2019

SOFTWARE SUPPORT FOR HARDWARE PREDICTORS

Mustafa Çavus
University of Rhode Island, mcavus@uri.edu

SOFTWARE SUPPORT FOR HARDWARE PREDICTORS

BY

MUSTAFA CAVUS

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2019

DOCTOR OF PHILOSOPHY DISSERTATION

OF

MUSTAFA CAVUS

APPROVED:

Dissertation Committee:

Major Professor Resit Sendag

Bin Li

Lutz Hamel

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2019

ABSTRACT

Hardware predictors are widely used to improve the performance of modern processors. These predictors are mostly used in data or instruction prefetching mechanisms and branch predictors. Hardware-based prefetchers and branch predictors can work dynamically based on the program's run-time behavior. However, most of the hardware-based predictor mechanisms depend on detecting patterns (data access patterns, branch patterns, etc) and they require very complex mechanisms to be able to capture irregular patterns.

Software techniques, like software prefetching, can help to improve the performance of the applications with behaviors that are difficult to capture by hardware mechanisms. On the other hand, they mostly rely on execution special instructions repeatedly during the execution which is likely to create an instruction overhead. Also, they cannot respond to the run-time behaviors like hardware mechanisms.

To overcome the weaknesses of prediction mechanisms, we proposed mechanisms which combine the strengths of hardware and software mechanisms. In this thesis, we examined ways to use the knowledge we can extract from the software to inform hardware mechanisms. We enable hardware-based systems to capture complex software behaviors just using the information it receives from the software instead of using large history tables and buffers to try to make predictions.

First, we proposed our hardware-based prefetching mechanism called Sequential Prefetcher with Adaptive Distance (SPAD). SPAD uses a simple method of hardware prefetching that integrates timeliness into sequential prefetching. It can outperform recently proposed complex prefetchers with simpler and smaller hardware.

In the second chapter, we proposed a software supported hardware prefetching mechanism called Array Tracking Prefether (ATP). This mechanism targets

irregular memory access patterns and relies on compiler/programmer to configure hardware prefetching mechanism. By combining the strength of software and hardware methods, ATP outperforms proposed software only and hardware-only solutions.

Finally, we proposed another software-assisted prefetcher for pointer intensive in-memory database applications, Node Tracker (NT). Although NT is proposed as a prefetcher, it is capable of using the knowledge it extracts from the prefetched data to help CPU pipeline in other ways to increase throughput. While tightly integrated with CPU, NT can achieve up to 19x speedup for the targeted applications.

ACKNOWLEDGMENTS

I am grateful to the many individuals who have helped me throughout my Ph.D. program. Firstly, I would like to express my sincere gratitude to my Major Professor, Prof. Resit Sendag, for the continuous support of my Ph.D. study and related research, his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I also would like to thank my committee members, Prof. Bin Li and Prof. Lutz Hamel for their support and encouragement. I also would like to thank Prof. Joan Peckham for her support as the additional committee member in my oral comprehensive exam and as the chair of my oral thesis defense.

I thank other faculty members of the Department of Electrical, Computer, and Biomedical Engineering, in particular to Prof. Jien-Chung Lo for his support during my teaching assistantship. I also would like to thank our graduate director Prof. Frederick J. Vetter, and our faculty staff members Meredith Leach Sanders and Lisa Pratt for their help during my study.

I would like to thank my family for standing by me and I would like to thank all my friends for their sincere support during my studies.

I also would like to thank my lab mate, Mohammed Shatnawi, for his support in the final stages of my research.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

Figure **Page**

LIST OF TABLES

CHAPTER 1

Introduction

Since the 1970s, microprocessor-based digital platforms have been riding Moores law, allowing for doubling of density for the same area roughly every two years. However, whereas microprocessor fabrication has focused on increasing instruction execution rate, memory fabrication technologies have focused primarily on an increase in capacity with negligible increase in speed. This divergent trend in performance between the processors and memory has led to a phenomenon referred to as the Memory Wall.

To overcome the memory wall, designers have resorted to a hierarchy of cache memory levels, which rely on the principle of memory access locality to reduce the observed memory access time and the performance gap between processors and memory. Unfortunately, important workload classes exhibit adverse memory access patterns that baffle the simple policies built into modern cache hierarchies to move instructions and data across cache levels. As such, processors often spend much time idling upon a demand fetch of memory blocks that miss in higher cache levels.

Prefetching−predicting future memory accesses and issuing requests for the corresponding memory blocks in advance of explicit accessesis an effective approach to hide memory access latency. There has been a myriad of proposed prefetching techniques, and nearly every modern processor includes some hardware prefetching mechanisms targeting simple and regular memory access patterns. In this chapter, we proposed an overview of the various classes of hardware prefetchers for instructions and data proposed in the research literature and presents examples of techniques incorporated into modern microprocessors.

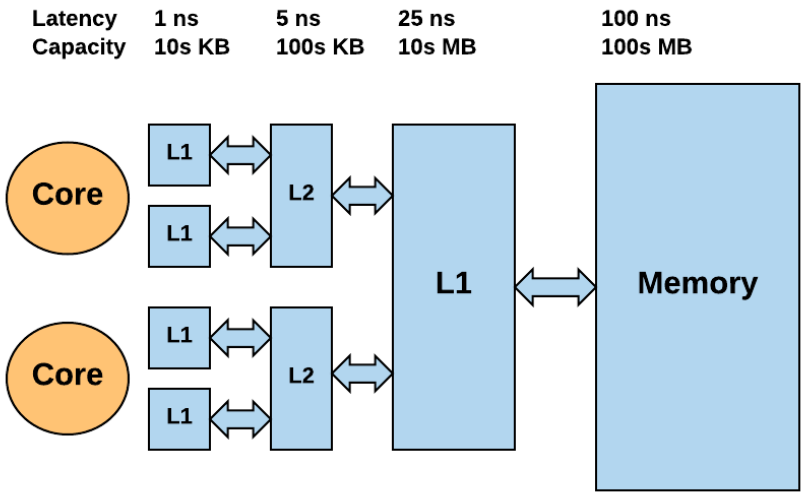


Figure 1.1: A modern memory hierarchy.

1.1 Memory Wall

Innovations in microarchitecture, circuits, and fabrication technologies have led to an exponential increase in processor performance over the past four decades. Meanwhile, DRAM has primarily benefitted from increases in density and DRAM speeds have improved only nominally. While future projections indicate that processor performance improvement may not continue at the same rate, the current gap in performance will necessitate techniques to mitigate long memory access latencies for years to come.

Computer architects have historically attempted to bridge this performance gap using a hierarchy of cache memories. Figure 1.1 depicts the anatomy of a modern computers cache hierarchy. The hierarchy consists of cache memories that trade-off capacity for lower latency at each level. The purpose of the hierarchy is to improve the apparent average memory access time by frequently handling a memory request at the cache, avoiding the comparatively long access latency of DRAM. The cache levels closer to the cores are smaller but faster. Each level provides a temporary repository for recently accessed memory blocks to reduce the

effective memory access latency. The more frequently memory blocks are found in levels closer to the cores, the lower the access latency. We refer to the cache(s) closest to the core as the L1 caches and then number cache levels successively, referring to the final cache as the last level cache (LLC).

The hierarchy relies on two types of memory reference locality. Temporal locality refers to memory that has been recently accessed and is likely to be accessed again. Spatial locality refers to memory in physical proximity that is likely to be accessed because near-neighbor instructions and data are often related.

While locality is extremely powerful as a concept to exploit and reduce the effective memory access latency, it relies on two basic premises that do not necessarily hold for all workloads, particularly as the cache hierarchies grow deeper. The first premise is that one cache size fits all workloads and access patterns. The capacity demands of modern workloads vary drastically, and differing workloads benefit from different trade-offs in the capacity and speed of cache hierarchy levels. The second premise is that a single strategy for allocating and replacing cache entries (typically allocating on-demand and replacing entries that have not been recently used) is suitable for all workloads. However, again, there is enormous variation in memory access patterns for which a simple strategy for deciding which blocks to cache may fare poorly.

There are a myriad of techniques that have been proposed from the algorithmic, compiler-level, and system software level all the way down to hardware to overcome the Memory Wall. These techniques include cache-oblivious algorithms, code and data layout optimizations at the compiler level, to hardware-centric approaches. Moreover, many software-based techniques have been proposed for prefetching.

1.2 Prefetching

One way to hide memory access latency is to prefetch. Prefetching refers to the act of predicting a subsequent memory access and fetching the required values ahead of the memory access to hide any potential long latency. In the limit, a memory access does not incur any additional overhead and memory appears to have a performance equal to a processor register. In practice, however, prefetching may not always be timely or accurate. Late or inaccurate prefetches waste energy and, in the worst case, can hurt performance.

To hide latency effectively, a prefetching mechanism must: (1) predict the address of a memory access (i.e., be accurate), (2) predict when to issue a prefetch (i.e., be timely), and (3) choose where to place prefetched data (and, potentially, which other data to replace).

1.2.1 Predicting Addresses

Predicting the correct memory addresses is a key challenge for prefetching mechanisms. If addresses are predicted correctly, the prefetching mechanism will have the opportunity to fetch them in advance and hide the memory access latency. If addresses are not predicted accurately, prefetching may cause pollution in the cache hierarchy (i.e., prefetched cache blocks would evict potentially useful cache blocks) and generate excessive traffic and contention in the memory system.

Predicting memory addresses may not be so simple. A data reference may be an access to a standalone variable or an element of a data structure and the nature of the reference depends on what the program is doing at a particular instance of execution. There are algorithms and data structure traversals that lend themselves well to both repetitive and predictable patterns (e.g., reading every element of an array sequentially). There are also a number of ways in which memory addresses can be hard to predict. These include, but are not limited

to, interleaving of accesses to variables, multiple data structures, and control-flow dependent traversals (e.g., searching a binary tree).

Similarly, an instruction reference will depend on whether the program is executing sequentially or it is taking a branch (i.e., following a discontinuity). While sequential instruction fetch is straightforward, the control-flow behavior and its predictability in the program can impact how effective instruction prefetching can be.

Predicting addresses accurately also depends on the level of the cache hierarchy at which the prefetching is performed. At the highest level, the interface between the processor and level-one cache (Figure 1.1) contains all memory reference information that could enable highly accurate prefetch, but could also lead to a waste of resources recording prefetch information for accesses that will hit in the first level cache anyway, and thus do not require prefetch. Conversely, at lower hierarchy levels, the access sequence is filtered, observing only the misses from higher levels. Thus, otherwise, effective prefetching algorithms may be confused by access-sequence perturbations from effects like cache placement and replacement policy.

Finally, there is typically a trade-off between the aggressiveness of a prefetch strategy and its accuracy; more aggressive prefetching will predict a higher fraction of the addresses actually requested by the processor at the cost of also fetching many more addresses erroneously. For this reason, many evaluation studies of prefetchers report two key metrics that jointly characterize the prefetchers effectiveness at predicting addresses. Coverage measures the fraction of explicit processor requests for which a prefetch is successful (i.e., the fraction of demand misses eliminated by prefetching). Accuracy measures the fraction of accesses issued by the prefetcher that turn out to be useful (i.e., the fraction of correct prefetches

over all prefetches). Many simple prefetchers can improve coverage at the expense of accuracy, whereas an ideal prefetcher provides both high accuracy and coverage.

1.2.2 Prefetch Lookahead

Ideally, a prefetching mechanism issues a prefetch well in advance and provides enough storage for prefetched data so as to hide all memory access latency. Predicting precisely when to prefetch in practice, however, is a major challenge. Even if addresses are predicted correctly, a prefetcher that issues prefetches too early may not be able to hold all prefetched memory close to the processor long enough prior to access. In the best case, prefetching too early will be useless because the prefetched information will be evicted away from the processor prior to use. In the worst case, it may evict other useful information (e.g., other prefetched memory or useful blocks in higher-level caches). If memory is prefetched late, then it will diminish the effectiveness of prefetching by exposing the memory access latency upon the memory access. In the limit, late prefetches may lead to performance degradation due to additional memory system traffic and poor interaction with mechanisms designed to prioritize time-critical demand accesses.

1.2.3 Placing Prefetched Values

The simplest and perhaps oldest software strategy for prefetching data is to load it into a processor register much like any other explicit load operation. Many architectures, in particular, modern out-of-order processors, do not stall execution when a load is issued, but rather stall dependent instructions only when the value of a load is consumed by another instruction. Such a prefetch strategy is often called a binding prefetch because the value of subsequent uses of the data is bound at the time the prefetch is issued. This approach comes with a number of drawbacks: (1) it consumes precious processor registers, (2) it obligates the hardware to perform

the prefetch, even if the memory system is heavily loaded, (3) it leads to semantic difficulties in the case the prefetch address is erroneous (e.g., should a prefetch of an invalid address result in a memory protection fault?), and (4) it is unclear how to apply this strategy to instructions.

Instead, most hardware prefetching techniques place prefetched values either directly into the cache hierarchy or into supplemental buffers that augment the cache hierarchy and are accessed concurrently. In multicore and multiprocessor systems, these caches and buffers participate in the cache coherence protocol, and hence the value of a prefetched memory location may change during the interval between the prefetch and a subsequent access; it is the hardwares responsibility to ensure the access sees the up-to-date value. Such prefetching strategies are referred to as non-binding. In these schemes, prefetching is purely a performance optimization and does not affect the semantics of a program.

1.3 Instruction Prefetching

Instruction fetch stalls are detrimental to performance for workloads with large instruction working sets; when the instruction supply slows down, the processor pipelines execution resources (no matter how abundant) will be wasted. Whereas desktop and scientific workloads often exhibit small instruction working sets, conventional server workloads and emerging cloud workloads exhibit primary instruction working sets often far beyond what upper-level caches can accommodate. With trends towards fast software development, scripting paradigms, and virtualized environments with increasing software stack depth, primary instruction working sets are also growing fast. Modern hardware instruction scheduling techniques, such as out-of-order execution, are often effective in hiding some or all of the stalls due to data accesses and other long latency instructions. However, out-of-order execution generally cannot hide instruction fetch latency. As

such, instruction stalls often account for a large fraction of overall memory stalls in servers.

Next-line prefetching [1] is the simplest form of instruction prefetching, which is prevalent in most modern processor designs. Because code is laid out sequentially in memory at consecutive memory addresses, often over half of the lookups in the instruction cache are for sequential addresses. The logic needed to generate sequential addresses and fetch them is minimal and fairly easy to incorporate into a processor and cache hierarchy.

1.4 Data Prefetching

Data miss patterns arise from the inherent structure that algorithms and high-level programming constructs impose to organize and traverse data in memory. Whereas instruction miss patterns in conventional von Neumann computer systems tend to be quite simple, following either sequential patterns or repetitive control transfers in a well-structured control flow graph, data access patterns can be far more diverse, particularly in pointer-linked data structures that enable multiple traversals.

Strided prefetchers are using a simple mechanism to identify unique strides that separates addresses in a memory stream based on the PC of the instructions that access them or based on global order [2]. Pointer chasing prefetchers, targeting to predict the address being pointed to by the pointers, try to predict future accesses by using hardware/software approaches. This can be achieved by inserting prefetch instructions via programmer/compiler [3] or by correlating the data in the data cache with its address and predicting its likelihood of being a pointer [4]. Irregular memory access pattern based prefetchers target harder to prefetch addresses by identifying certain key characteristics of the memory stream [5, 6, 7, 8]. Markov prefetchers [6, 9, 10] predict from an observed memory sequence, the sets

of unique addresses that are likely to occur in the future.

1.4.1 Stream and Stride Prefetchers

The first category of data prefetchers is stride and stream prefetchers, which are a direct evolution of the next-line and stream prefetching mechanisms that have been mainly developed for instructions. These prefetchers capture access patterns for data that are either laid out contiguously in the virtual address space or are separated by a constant stride. This class of prefetcher tends to be highly effective for dense matrix and array access patterns, but generally provides little benefit for pointer-based data structures. Strided data prefetchers are widely deployed in industrial processor designs, from systems as old as the IBM System/370 series through modern high-performance processors. Until recently, it is believed that this class of hardware data prefetcher was the only class to be commercially deployed.

Sequential data prefetcher implementations, which are restricted to prefetch only blocks at consecutive addresses, were described as early as 1978 [1]. By the early 1990s such prefetchers were extended to detect and prefetch sequences of accesses separated by a non-constant stride [11]. Such strided access patterns arise frequently when traversing multi-dimensional arrays or when aggregate data types (e.g., structs in C) are stored in arrays. Strided accesses can also arise by happenstance even in pointer-based data structures when dynamic memory allocators layout constant-sized objects consecutively in memory, a common case due to pool allocators. Dahlgren and Stenstrom study the relative merits and effectiveness of sequential and stride prefetching mechanisms in detail [12].

A key challenge in stride prefetcher implementations is to distinguish among multiple inter-leaved strided sequences, for example, as may arise in a matrix-vector product. Figure 1.2 shows a diagram of Baer and Chens scheme to track

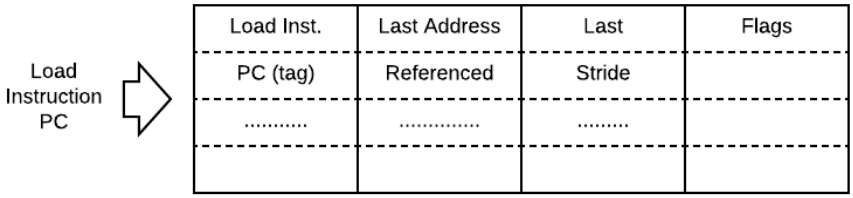


Figure 1.2: A diagram of Baer and Chens prediction table [11].

strides on a per-load-instruction basis. Their reference prediction table is a tagged, set-associative structure that uses the load instruction PC as the lookup key. Each entry holds the last address referenced by that load and the difference in address (i.e., stride) between the last two preceding references. Whenever the same stride is observed twice consecutively, the last reference address and stride are used to compute one or more additional addresses for prefetch. Subsequent accesses that continue to match the recorded stride will trigger additional prefetches. A long sequence of such strided accesses is referred to as a stream, analogous to instruction stream prefetchers. Ishii and co-authors describe more sophisticated hardware structures that can compactly represent multiple strides [13], while Sair and co-authors extend stream prefetching to more irregular patterns by predicting stride lengths [14].

A second key implementation issue is to decide how many blocks to prefetch when a strided stream is detected. This parameter, often referred to as the prefetch degree or prefetch depth, is ideally large enough that the prefetched data arrive before being referenced by the processor, but not so large that blocks are replaced before access or cause undue pollution for short streams. Hur and Lin propose simple state machines that track histograms of recent stream lengths and can adaptively determine the appropriate prefetch depth for each distinct stream, enabling stream prefetchers to be effective even for short streams of only a few addresses [15].

Conventionally, stride prefetchers place the data they fetch directly into the cache hierarchy. However, if stride prefetchers are aggressive, they may pollute the cache, displacing useful data. Jouppi [16] describes an alternative organization wherein stream prefetchers place data in separate buffers, called stream buffers, which are accessed immediately after or in parallel with the L1 cache. By placing data in a stream buffer, a low accuracy stream (where many data are fetched but not used) does not displace useful data in the cache, reducing the risk of inaccurate prefetching. How-ever, erroneous prefetches still consume energy and bandwidth. Palacharla and Kessler evaluate a memory system organization where stream buffers entirely replace the second-level data cache [2].

Each stream buffer holds cache blocks from a single stream. Accesses from the processor interrogate the stream buffer contents, typically in parallel with accesses to the L1 cache. A hit in a stream buffer typically causes the requested block to be transferred to the L1 cache and an additional block from the stream to be fetched. In some variants, stream buffers are strictly FIFO and only the head of each stream buffer may be accessed. In other variants, stream buffers are associatively searched. When the stride detection mechanism observes a new stream, an entire stream buffer is cleared and re-allocated (discarding any unreferenced blocks from a stale stream), typically according to a round-robin or least-recently-used scheme.

1.4.2 Address-Correlating Prefetchers

Whereas stride prefetchers are typically ineffective for pointer-based data structures, such as linked lists, the second class of prefetcher we consider is specifically designed to target the pointer-chasing access patterns of such data structures. Instead of relying on regularity in the layout of data in memory, this class of prefetcher exploits the fact that algorithms tend to traverse data structures in the same way repeatedly, leading to recurring cache miss sequences.

Correlation between accesses to pairs of memory locations was suggested as early as 1976 [17]. Charney and Reeves first described hardware prefetchers that seek to exploit such pair-wise correlation relationships, coining the term "correlation-based prefetcher" [18, 19]. Later work generalizes the notion of address correlation from pairs to groups or sequences of accesses [20, 21]. Wenisch and co-authors introduce the term "temporal correlation" [21] to refer to the phenomenon that two addresses accessed near one another in time will tend to be accessed together again in the future. Temporal correlation is an analog to "temporal locality", that a recently accessed address is likely to be accessed again in the near future. Whereas caches exploit temporal locality, address-correlating prefetchers exploit temporal correlation.

Jump Pointers

Correlating prefetchers are a generalization of hardware and software mechanisms that specifically targeted pointer-chasing access patterns. These earlier mechanisms rely on the concept of a jump pointer [22, 23, 24, 25], a pointer that enables a large forward jump in a data structure traversal. For example, a node in a linked list may be augmented with a pointer ten nodes forward in the list; the prefetcher can follow the jump pointer to gain lookahead over the main traversal being carried out by the CPU, enabling timely prefetch. Prefetchers relying on jump pointers often require software or compiler support to annotate pointers. Content directed prefetchers [26, 27] eschew annotation and attempt instead to dereference and prefetch any load value that appears to form a valid virtual address. While jump-pointer mechanisms can be quite effective for specific data structure traversals (e.g., linked list traversals), their key shortcoming is that the distance the jump pointer advances the traversal must be carefully balanced to provide sufficient lookahead without jumping over too many elements. Jump pointer

distances are difficult to tune and the pointers themselves can be expensive to store.

Pair-wise Correlation

In essence, a correlation-based hardware prefetcher is a lookup table that maps from one address to another address that is likely to follow it in the access sequence. While such an association can capture sequential and stride relationships, it is far more general, capturing, for example, the relationship between the address of a pointer and the address to which it points. It is the ability to capture pointer traversals that affords address-correlating prefetchers a far greater opportunity for performance improvement than stride prefetchers, as pointer-chasing access patterns are disproportionately slow on modern processors. However, address-correlating prefetchers rely on repetition; they are unable to prefetch addresses that have never previously been referenced (in contrast to stride prefetchers). Moreover, address correlation prefetchers require enormous state, as they need to store the successor for every address. Hence, their storage requirement grows proportionally to the working set of the application. Much of the innovation in address-correlating prefetcher design centers on managing this enormous state.

Markov Prefetcher

The Markov prefetcher [28, 29] is the simplest prefetcher design to exploit pair-wise address correlation. It directly implements the notion of a look-up table mapping a trigger address to its immediate successor in the off-chip access sequence. However, because addressesespecially when considered at cache-block granularityoften participate in multiple traversals, storing only a single successor for each trigger address results in poor effectiveness. Instead, the Markov prefetcher stores several previously observed successors, all of which are prefetched when a

miss to the trigger address is observed. By prefetching several possible successors, the Markov prefetcher sacrifices accuracy (fraction of correct prefetches over all prefetches) to improve coverage (fraction of demand misses for which a prefetch is successful)only one of the addresses requested by the prefetcher is expected to be correct. The number of successors fetched is often referred to as the width of the prefetch.

Global History Buffer

The cache-like organization of a Markov prefetcher limits it to record only fixed-length streamsa single table entry can store addresses for only a fixed prefetch depth. Narrow table entries sacrifice potential coverage and lookahead, while wide table entries are storage-inefficient for short streams. Entries can be chained together via pointers, however, this increases lookup latency and is particularly undesirable if correlation tables are located off-chip. Wenisch and co-authors study repetitive temporally correlated streams in commercial server applications and demonstrate that stream lengths vary from two to many thousands of cache blocks [30, 31]. The most common stream length is only two misses, implying that a wide Markov table entry is storage inefficient. However, when weighted by the number of misses in the stream (i.e., the potential coverage that can be obtained by prefetching the stream), the median stream length is about ten cache blocks.

A key advance, introduced by Nesbit and Smith in their global history buffer [32], is to split the correlation table into two structures: a history buffer, which logs the sequence of misses in a circular buffer in the order they occurred, and an index table, which provides a mapping from an address (or other prefetch trigger) to a location in the history buffer. The history buffer allows a single prefetch trigger to point to a stream of arbitrary length. Figure 1.3 illustrates the Global History Buffer organization.

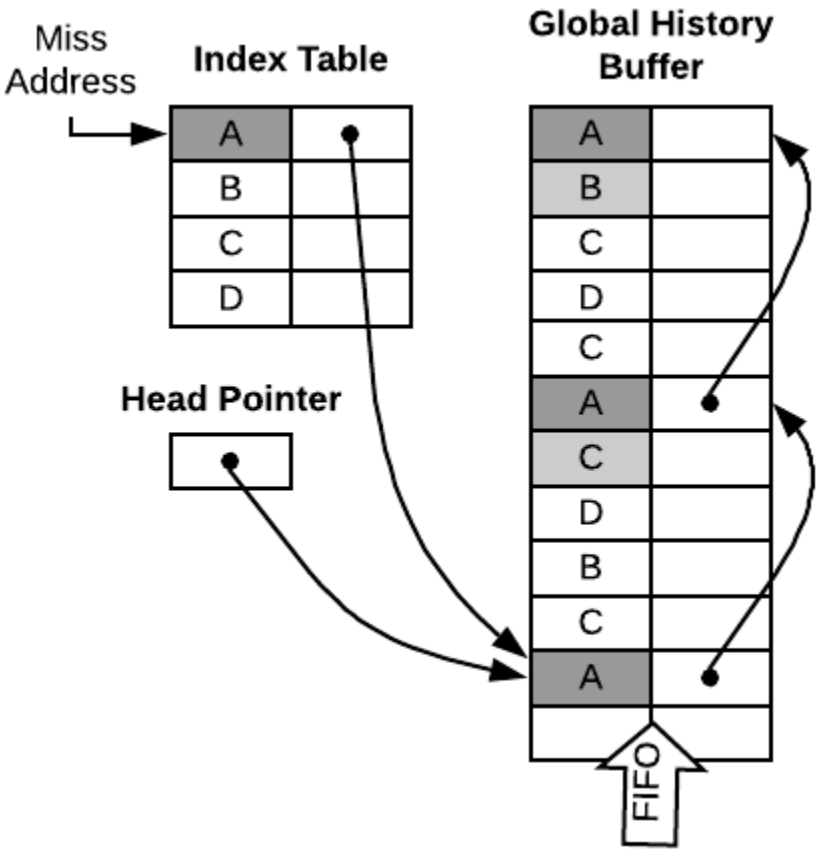


Figure 1.3: Address-correlating global history buffer (GHB G/AC) [32].

The index table retains a set-associative storage organization similar to the original Markov prefetcher. However, rather than storing cache block addresses, the index table now stores pointers into the history buffer. When a miss occurs, the GHB references the index table to see if any information is associated with the miss address. If an entry is found, the pointer is followed and the history buffer entry is checked to see if it still contains the miss address (the entry may since have been overwritten). If so, the next few entries in the history buffer contain the predicted stream. History buffer entries can also be augmented with link pointers to other history buffer locations, to enable history traversal according to more than one ordering (e.g., each link pointer may indicate a preceding occurrence of the same miss address, enabling increases to prefetch width as well as depth).

1.4.3 Execution-Based Prefetching

Another category of data prefetcher relies neither on repetition in miss sequences nor in data layouts; rather execution-based prefetchers seek to explore the programs instruction sequence ahead of instruction execution and retirement to discover address calculations and dereference pointers. The key objective of such prefetchers is to run faster than instruction execution itself, to get ahead of the processor core, while still using the actual address calculation algorithm to identify prefetch candidates. As such, these mechanisms do not rely on repetition at all. Instead, they rely on mechanisms that either summarize address calculation while omitting other aspects of the computation, guess at values directly, or leverage stall cycles and idle processor resources to explore ahead of instruction retirement.

Algorithm Summarization

Several prefetching techniques summarize the instruction sequence that traverses a data structure, such that the traversal pattern can be executed faster than the main thread to prefetch data structure elements. Roth and co-authors [3, 24] propose a mechanism that summarizes traversals entirely in hardware by identifying pointer loads (load instructions that dereference a pointer) and the dependent chain of instructions that connect them. These dependence relationships are then encoded by hardware into a compact state machine, which can iterate through the sequence of dependent loads faster than instruction execution. Annavaram, Patel, and Davidson propose a general mechanism for extracting program dependence graphsa subset of instructions that lead to missing loadsin hardware and then executing these graphs in dedicated precomputation engines [33].

Helper-Thread and Helper-Core Approaches

Thread-based data prefetching techniques [34, 35, 36, 37, 38, 39, 40, 41, 42] use idle contexts on a multithreaded or multicore processor to run helper threads that overlap misses with speculative execution. Individual techniques vary in whether they are automatic or require compiler/software support, whether they rely on simultaneous multithreading hardware and specific thread coordination mechanisms, whether they rely on additional cores, and whether they require additional mechanisms to insert blocks into remote caches. In nearly all cases, these techniques re-purpose spare execution contexts to execute the prefetching code. However, the spare resources the helper threads require (e.g., idle cores or thread contexts; fetch and execution bandwidth) may not be available when the processor executes an application exhibiting high thread-level parallelism. The benefit of these techniques must be weighed against scaling up the number of application threads.

Run-Ahead Execution

Run-ahead execution uses the execution resources of a core that would otherwise be stalled on a long-latency event (e.g., an off-chip cache miss) to explore ahead of the stalled execution in an effort to discover additional load misses and warm branch predictors. The idea in run-ahead is to capture a snapshot of execution state when the core would otherwise stall, then proceed past stalled instructions to continue to fetch and execute the predicted instruction stream. Instructions that are data-dependent on an incomplete instruction are not executed (e.g., a poison token is propagated through the register renaming mechanism). When the long-latency event resolves (e.g., the original miss returns), the execution state is recovered from the snapshot and the original execution continues, re-crossing the instructions that were explored during run-ahead. The primary benefit of this scheme is the prefetching effect for long-latency loads. Run-ahead

Code Snippet 1.1: A simple example code to demonstrate software prefetching.

```
for (i = 0; i < N; i++) {
    prefetch(&(A[i+1]));
    sum += A[i];
}
```

was originally proposed in the context of in-order cores by Dundas and Mudge [43]. Mutlu and co-authors explore efficient implementations in the context of out-of-order processors [44, 45, 46, 47]. More recently, authors have explored non-blocking pipeline microarchitectures that speculate past long-latency loads without discarding speculative execution results when the loads return, instead re-executing only the dependent instructions [48, 49].

1.4.4 Software prefetching

With software prefetching the programmer or compiler inserts prefetch instructions into the program. These are instructions that initiate a load of a cache line into the cache but do not stall waiting for the data to arrive. Code Snippet 1.1 demonstrates a simple example code with software prefetching.

Processors that have multiple levels of caches often have different prefetch instructions for prefetching data into different cache levels. This can be used, for example, to prefetch data from main memory to the L2 cache far ahead of the use with an L2 prefetch instruction, and then prefetch data from the L2 cache to the L1 cache just before the use with an L1 prefetch instruction.

There is a cost for executing a prefetch instruction. The instruction has to be decoded and it uses some execution resources. A prefetch instruction that always prefetches cache lines that are already in the cache will consume execution resources without providing any benefit. It is therefore important to verify those prefetch instructions prefetch data that is not already in the cache.

The cache miss ratio needed by a prefetch instruction to be useful depends on its purpose. A prefetch instruction that fetches data from main memory only needs a very low miss ratio to be useful because of the high main memory access latency. A prefetch instruction that fetches cache lines from a cache further from the processor to a cache closer to the processor may need a miss ratio of a few percent to do any good.

Commonly, software prefetching creates fetches slightly more data than actually used. For example, when iterating over a large array it is common to prefetch data some distance ahead of the loop. When the loop is approaching the end of the array the software prefetching should ideally stop. However, it is often cheaper to continue to prefetch data beyond the end of the array than to insert additional code to check when the end of the array is reached. This means that 1 kilobyte of data beyond the end of the array that isn't needed is fetched.

1.5 Recent Work in Prefetching

1.5.1 Sandbox Prefething

Sandbox Prefetching (SBP) [50] works by testing out several aggressive sequential prefetchers in a sandboxed environment outside the real memory hierarchy in order to determine which prefetchers should be used in the real memory hierarchy. Rather than issuing real prefetches, SBP evaluates prefetchers by placing prefetch addresses in a Bloom filter which is a data structure designed to tell you, rapidly, whether an element is present in a set. Demand cache accesses check the Bloom filter to see if the address could have been prefetched by the prefetcher currently being evaluated. Hits in the Bloom filter give confidence that the evaluated prefetcher would be accurate if it were deployed in the real memory hierarchy. Several prefetchers are evaluated in a round-robin fashion, and the prefetchers with the most Bloom filter hits are used to issue real prefetches. SBP evaluates sequen-

tial aggressive prefetchers that immediately prefetch addresses with a fixed offset from the current demand access, like a next-line prefetcher. Once deployed in the real memory hierarchy, the chosen prefetchers perform no additional warm-up or confirmation before issuing prefetches.

1.5.2 Indirect Memory Prefetcher

Indirect Memory Prefetcher (IMP) [51] is designed to capture indirect memory accesses. Indirect memory accesses are mostly in the form of multiple arrays where the values of an array are used as an index to access another array. For example, in an A[B[i]] structure where A and B are arrays in an application, the values of array B are needed to access array A. Initially, IMP works like a stride prefetcher to detect sequential accesses to an array (which is the accesses to array B in A[B[i]]). Then, using additional components, it tries to detect if any of the other accesses are related to the values of detected sequential access (which is the accesses to array A in A[B[i]]). This is a pure hardware approach to detect indirect accesses automatically and issue prefetches for them.

1.5.3 Software Prefetching for Indirect Memory Accesses

Ainsworth proposed a compiler pass to automatically generate software prefetching instructions for indirect memory accesses [52]. Within the compiler, it finds the loads that reference loop induction variable, and use a depth-first search algorithm to identify the set of instruction which needs to be duplicated to load in data for future iterations. Across the different workloads they evaluated, they achieved an average of 1.3x performance improvement for an Intel Haswell machine, 1.1x for an ARM Cortex-A57, 2.1x for an ARM Cortex-A53, and 2.7x for a Xeon Phi.

1.5.4 An Event-Triggered Programmable Prefetcher for Irregular Workloads

Ainsworth [53] proposed a software-assisted hardware prefetching mechanism to prefetch irregular access patterns. It employs low power RISC cores to execute subprograms to compute future addresses and prefetch them. It relies on the programmer or compiler to generate the subprograms to generate subprograms. Prefetch calculations are triggered by events which are either load accesses to the cache or cache fills with the prefetched data. Each event triggers an execution on one of the available cores to calculate and issue prefetches. It captures a variety of irregular access behaviors like pointer-chasing memory accesses and indirect memory accesses.

1.6 Overview of the Research Topics Covered in This Thesis

1.6.1 Time-Effective Sequential Prefetching

Sequential prefetching is a widely employed and useful technique to capture regular memory access patterns. However, without being timely accurate, they are unable to hide memory access latencies sufficiently. To be timely accurate, sequential prefetchers should use a prefetch distance which is how far ahead the prefetches should be issued. If the distance is too long, prefetches might be too early which may cause the prefetched data to be evicted from the cache before it is used. On the other hand, if the distance is too short, prefetches end up to be too late that they cannot hide the access latency well. In our experiments, we observe that the ideal distance varies by different application behaviors. So, employing a dynamic mechanism to adjust the prefetch distance improves the overall benefit of a sequential prefetcher significantly. In this work, we explained our adaptive distance adjusting mechanism for sequential prefetchers.

1.6.2 Informed Prefetching for Indirect Memory Accesses

Indirect memory accesses (such as A[B[i]] where array B is accessed sequentially) are common in applications involving graph-based data structures, sparse matrices, etc. These applications tend to have irregular access patterns which are very hard to be predicted by pure hardware mechanisms and existing methods either require very expensive and complex mechanisms or can capture only limited types of behaviors. Software-based prefetching techniques can easily prefetch these accesses but they have the disadvantage of having instruction overhead due to the extra instructions related to prefetching. In this study, we developed a software-assisted hardware mechanism, Array Tracking Prefetcher, to prefetch indirect memory accesses.

1.6.3 Informed Pre-Execution for In-Memory Database Applications

Pointer-chasing access behaviors are common in in-memory database applications. These algorithms commonly include multiple lookups over a pointer-intensive data structure where each lookup iterates over a set of nodes, and each node has one or multiple pointers which points to the next node to be accessed. As well as the huge number of cache misses created by these applications, long dependency chains also create an important performance bottleneck for these applications. Processing multiple lookups in parallel improves the performance of these applications significantly. However, hiding memory access latencies still, do not maximize the throughput of these applications. Due to long dependency chains and high branch misprediction rates, prefetching only solutions have limited potential. In this work, we proposed Node Tracker, a software/hardware system, which prefetches/pre-executes future lookups in parallel, and further improves the CPU performance by using the knowledge which is extracted from the pre-executions.

List of References

[1] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec 1978.

[2] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

[3] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 115–126, 1998.

[4] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *ACM SIGPLAN Notices*, vol. 37, no. 10. ACM, 2002, pp. 279–290.

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Stealth prefetching," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 274–282.

[6] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *ACM SIGPLAN Notices*, vol. 37, no. 5. ACM, 2002, pp. 199–209.

[7] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

[8] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 252–263.

[9] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 252–263.

[10] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 222–233, 2005.

[11] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 176–186. [Online]. Available: http://doi.acm.org/10.1145/125826.125932

[12] F. Dahlgren and P. Stenstrom, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, Jan 1995, pp. 68–77.

[13] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 499–500. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542349

[14] S. Sair, T. Sherwood, and B. Calder, "A decoupled predictor-directed stream prefetching architecture," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 260–276, March 2003.

[15] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 397–408.

[16] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 364–373. [Online]. Available: http://doi.acm.org/10.1145/325164.325162

[17] J. L. Baier and G. R. Sager, "Dynamic improvement of locality in virtual memory systems," *IEEE Trans. Softw. Eng.*, vol. 2, no. 1, pp. 54–62, Jan. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233801

[18] M. J. Charney and A. P. Reeves, "Generalized correlation based hardware prefetching," 1995.

[19] M. J. Charney, "Correlation-based hardware prefetching," Ph.D. dissertation, Ithaca, NY, USA, 1995, uMI Order No. GAX95-42429.

[20] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," *SIGPLAN Not.*, vol. 37, no. 5, pp. 199–209, May 2002. [Online]. Available: http://doi.acm.org/10.1145/543552.512554

[21] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 222–233, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069989

[22] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 222–233, Sept. 1996. [Online]. Available: http://doi.acm.org/10.1145/248208.237190

[23] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 115–126, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/384265.291034

[24] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 111–121.

[25] J. D. Collins, S. Sair, B. Calder, and D. Tullsen, "Pointer cache assisted prefetching," vol. 2002, 01 2002, pp. 62–73.

[26] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 279–290, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635506.605427

[27] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 7–17.

[28] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 252–263, May 1997. [Online]. Available: http://doi.acm.org/10.1145/384286.264207

[29] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 121–133, Feb. 1999. [Online]. Available: https://doi.org/10.1109/12.752653

[30] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 99–108.

[31] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 79–90.

[32] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, Feb 2004, pp. 96–96.

[33] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 52–61.

[34] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 14–25, May 2001. [Online]. Available: http://doi.acm.org/10.1145/384285.379248

[35] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 306–317.

[36] I. Ganusov and M. Burtscher, "Future execution: A hardware prefetching technique for chip multiprocessors," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 350–360. [Online]. Available: http://dx.doi.org/10.1109/PACT.2005.23

[37] I. Ganusov and M. Burtscher, "Future execution: A prefetching mechanism that uses multiple cores to speed up single threads," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 4, pp. 424–449, Dec. 2006. [Online]. Available: http://doi.acm.org/10.1145/1187976.1187979

[38] Y. Solihin, C. Jung, D. Lim, and J. Lee, "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 09, pp. 1309–1324, sep 2009.

[39] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–95. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346187

[40] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Microarchitectural support for precomputation microthreads," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 74–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=774861.774870

[41] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," *SIGPLAN Not.*, vol. 46, no. 3, pp. 393–404, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950411

[42] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001, pp. 37–48.

[43] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 68–75. [Online]. Available: http://doi.acm.org/10.1145/263580.263597

[44] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, Feb 2003, pp. 129–140.

[45] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, Nov 2003.

[46] Onur Mutlu, Hyesoon Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, June 2005, pp. 370–381.

[47] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *Micro, IEEE*, vol. 26, pp. 10 – 20, 02 2006.

[48] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *SIGOPS Oper. Syst. Rev.*, vol. 38, no. 5, pp. 107–119, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1037949.1024407

[49] A. Hilton, S. Nagarakatte, and A. Roth, "icfp: Tolerating all-level cache misses in in-order processors," *IEEE Micro*, vol. 30, no. 1, pp. 12–19, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.20

[50] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 626–637.

[51] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 178–190.

[52] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 305–317.

[53] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," *SIGPLAN Not.*, vol. 53, no. 2, pp. 578–592, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3296957.3173189

CHAPTER 2

Time-Effective Sequential Prefetching

Mustafa Cavus[1], Resit Sendag[2]

[1,2]Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI02882.

2.1 Abstract

Current processors employ aggressive hardware prefetching mechanisms to improve performance and reduce power. Sequential prefetching is a widely employed and successful technique that exploits spatio-temporal memory access patterns in applications. However, it does not take into account prefetch timeliness. We propose a simple method that integrates timeliness into sequential prefetching. Our results show that 139-bytes direct-mapped mechanism can significantly improve the performance of an L2 sequential prefetcher and can match or outperform recently proposed complex prefetchers with simpler and smaller hardware.

2.2 Introduction

Modern processors employ prefetchers to hide long memory latencies for demand cache misses. Prefetchers predict data or instruction addresses those are likely to be used in the near future. When successful, they facilitate faster retrieval of data/instruction for demand requests. Next-line or sequential prefetching has been shown to provide significant performance benefits for applications with a good spatial locality. However, they prefetch rather blindly because they do not employ confidence mechanisms. This is problematic for two reasons: 1) they use cache and bandwidth resources rather blindly, which can either reduce their benefit, or can even hurt the performance and power; 2) even for the applications with good spatial locality, they may not provide the potential benefits because they are not timely in issuing prefetches. To address the first problem, Pugsley et al. [1] proposed the sandbox prefetching (SBP) method. In their method, a set of predetermined sequential offset prefetchers (hence, it is also called offset prefetching) are tested by recording their predicted prefetching addresses in the sandbox on each demand access and counting the number of demand access hits on the recorded potential prefetch addresses in the sandbox. After the evaluation interval, only the

offsets with sandbox scores above a threshold are allowed to perform prefetching in the next interval. The sandbox proves to be a powerful idea eliminating many unnecessary and potentially harmful prefetches only after a prefetcher has been proven useful, it is activated.

The second problem, although equally important in designing successful prefetchers, is not sufficiently addressed by the SBP. If prefetch is not timely, there is no benefit. Best Offset Prefetcher (BOP) [2], which was the winner of the 2015 Data Prefetching Competition [3], develops a method to target prefetch timeliness for SBP. Similar to SBP, various offsets compete in a history table (e.g., Sandbox) and the best performing offset is chosen to perform prefetching in the next interval. However, the decision for the best offset is made by not the only number of correct predictions but also their timeliness. In order to track timeliness, BOP records the time, i.e, cycle, at which a prefetched cache line is placed in the cache. That is, it records the time of the cache line refill. This requires BOP to employ one bit per L2 cache line to track prefetched lines in the cache and observe their refill times to store in an auxiliary table to determine timeliness. In this work, we focus on both timeliness and accuracy, as in the BOP. Instead of offset-testing via a sandbox approach, however, we focus on the most popular offset, +1 (i.e., next-line), that occurs in most applications, and we propose a simple mechanism to guide the sequential prefetcher for timeliness. By dynamically adapting distance (hence, we call it Sequential Prefetcher with Adaptive Distance (SPAD)) [4] for the sequential prefetcher, we show that our proposed mechanism outperforms the BOP, with less hardware and lower complexity.

SPAD uses a testing queue, TQ, in the same spirit as the SBP method but its operation and purpose are quite different (as described in Section 2.5). After each evaluation period, SPAD's decision engine increments or decrements a distance

counter to guide the sequential prefetcher in how far ahead a prefetch must be issued in the next interval to be useful. The decision on incrementing or decrementing the distance is based on several factors, such as, the number of demand hits in the TQ, the number of L2 misses and the amount and ratio of demand misses that are hit in the TQ. It is important to note that SPAD actively issues prefetches and gets evaluated at the same time using only one single testing buffer. In addition, SPAD does not need to keep track of the prefetched lines and their refill times in the cache and therefore, despite providing better performance than BOP, it uses much simpler logic and much less hardware storage.

This chapter makes the following contributions:

1. It presents a detailed analysis of offset prefetching and provides insights into the understanding of offset prefetching performance.

2. It shows that although best performing offset values are larger than 1, these offset values are rarely observed delta values in SPEC CPU2006 benchmarks. Most performance benefit, in fact, comes from the most frequently observed address delta value 1, but prefetching is more timely with offsets larger than 1.

3. It categorizes SPEC CPU2006 benchmarks based on their offset prefetching and delta pattern behaviors.

4. It proposes a simple and highly effective algorithm to track prefetch timeliness focusing on delta 1 prefetching. The proposed SPAD prefetcher outperforms recent offset prefetchers, SBP and BOP, with significantly lower hardware budget.

The rest of the chapter is organized as follows. Section 2.3 discusses related work. Section 2.4 presents a detailed analysis of offset prefetching and motivates

our work. Section 2.5 describes the proposed SPAD prefetcher. Methodology is given in Section 2.6. In Section 2.7, we present the results. Finally, Section 2.8 concludes.

2.3 Background

We evaluate SPAD by comparing it to 5 state-of-the-art hardware prefetchers: Sandbox Prefetching (SBP) [1], Best Offset Prefetcher (BOP) [2], Access Map Pattern Matching (AMPM) [5], Global History Buffer (GHB) [6] and Variable Length Delta Prefetching (VLDP) [7]. All these prefetchers exploit regular spatio-temporal patterns similar to SPAD.

There are many earlier prefetchers that focus on regular patterns deserves a mention. The earliest of all is the nextline prefetching [8]. To avoid issuing useless prefetches, a prefetch bit can be added to each cache line [8] or cache replacement status [9] can be used instead. Stride prefetchers are confidence-based prefetchers which exploit constant strides among the instructions which have memory access [10, 11, 12]. To exploit sequential streams, stream buffer was introduced by Jouppi [13]. A stream buffer that can also track non-unit stride accesses was later proposed by Palacharla and Kessler [14]. Other work [15, 16, 17] studied more aggressive stream buffers that exploit adaptive degree and distance values. Finally, some prefetchers used history tables to record and monitor the past memory accesses to predict future addresses [6, 5, 18, 19, 20, 21, 22, 23, 24]. In this chapter, we compare our proposal with more recent work that outperform earlier prefetchers. In this section, we describe them in detail.

In our evaluation, all prefetchers live at the L2 cache level. The information available to the prefetcher at this level of cache hierarchy is limited. In most current processors, the program counter (PC) is unavailable at this level. This makes PC-based patterns harder to track. Furthermore, a prefetcher at this level

must deal with physical addresses directly without the TLB or other page table information. Therefore, many prefetchers track addresses on a per physical page basis, discovering patterns and prefetching within multiple simultaneously tracked physical pages. This complicates the effective design for the prefetcher, especially in terms of efficiently tracking multiple simultaneous physical pages. All prefetchers that we evaluated in this work were originally proposed as L2 prefetchers, except for GHB [6].

2.3.1 Sandbox Prefetching

Sandbox Prefetching (SBP) is the first full-fledged offset prefetcher. It is cost-effective and was shown to slightly outperform the more complex AMPM prefetcher [5] that won the 2009 Data Prefetching Competition [25]. The SBP works by evaluating and scoring candidate prefetchers without issuing actual prefetch requests. Instead, these candidate requests are recorded in a Bloom filter structure. The accuracy of these predicted prefetch addresses is evaluated by checking if subsequent demand accesses hit in the Bloom filter. Pugsley et al. proposed offset prefetchers as candidate prefetchers. A number of fixed-offset prefetchers (that prefetch $X + O$, where X is the block address requested and O is the non-zero offset) are evaluated using a sandbox and the ones with a score above a threshold are allowed to issue prefetch requests until the next evaluation period has been completed and new scores have been obtained. The score for an offset is simply the number of hits in the sandbox (i.e., the Bloom filter) during the interval where that offset has been tested.

SPAD and SBP are both sequential prefetchers. The SBP does not take into account prefetch timeliness when evaluating offsets and therefore its scoring mechanism is suboptimal. Offsets are simply tested for accuracy by counting the number of hits in the sandbox. SPAD exploits spatio-temporal patterns of one

single offset and uses a sandbox-like table to test both prefetch timeliness and accuracy.

2.3.2 Best Offset Prefetcher

The Best Offset Prefetcher (BOP), which is the winner of the 2015 Data Prefetching Competition, is an offset prefetcher like the SBP. It stores in a Recent Request (RR) table the base addresses of each recent prefetch, that is, the address of the demand access that generated the prefetch. On every L2 miss or prefetched hit for a block address X, BOP tests n^{th} offset O_n from the offset list, by searching if the block address $X - O_n$ is in the RR table. If it is found in the RR table, it means that block X would likely have been prefetched successfully with offset O_n and the score for offset O_n is incremented. Next L2 access tests offset O_{n+1} and so on until a full round is finished (i.e., all the offsets has been tested once). Then, n is reset and a new round starts. After a predetermined number of rounds (e.g., 100) have been completed, the BOP chooses the best offset to issue prefetches with, i.e., the offset with the highest score. Then, the scores and the round counter are reset and a new learning phase starts.

Similar to SPAD, BOP considers both accuracy and timeliness of prefetches. However, SPAD is not an offset prefetcher because it only considers one single offset. Our results show that by focusing on most commonly observed offset 1 (i.e., the resulting prefetcher is a next-line prefetcher) and the timeliness of the issued prefetches, SPAD outperforms BOP with less hardware and lower complexity.

2.3.3 Access Map Pattern Matching (AMPM)

AMPM [5] works by tracking the status of each cache line (untouched, demand accessed, prefetched) within large regions of memory (e.g., 4KB in our evaluation, i.e., one page). It uses two-bit vectors (one to track demand accesses and the

other to track prefetches) where each bit in a vector representing a cache line. The access map for a region starts with the bit vectors zeroed out and then marks each cache line that has been demand accessed. For a cache access to address X, the determination whether $X + O$ should be prefetched is done by checking the access map to see if $X - O$ and $X - 2O$ have both been accessed before. This is done for a number of O values. Prefetch aggressiveness could be controlled statically or by a dynamic feedback mechanism. Preference is given to prefetch addresses closer to X.

AMPM exploits spatio-temporal access patterns and performs quite well. However, per-page tracking of each cache line requires much larger hardware storage than SPAD to perform similarly. Another disadvantage of AMPM-like prefetcher is its training time. AMPM requires three accesses along a stride pattern within a region before prefetching starts. This warm-up is needed for each region independently. SPAD, being a sequential prefetcher, does not need this warm-up or confirmation before issuing prefetches. Finally, unlike SPAD, AMPM does not consider the timeliness of the issued prefetches.

2.3.4 Global History Buffer

Global History Buffer (GHB) [6] provides a framework for implementing various prefetcher mechanisms. In this work, we evaluate the GHB PC/DC prefetcher. GHB employs two main components: the index table and the global history buffer (GHB). The index table is indexed by the PC and the GHB is indexed by pointers in the index table. Each index table entry points to the most recent delta of that PC in the global history. Each entry in the GHB stores the last delta and a pointer to the next instance of the current PC. The links in the GHB can be followed to replay the deltas that were seen previously.

GHB PC/DC is the only L2 prefetcher that we studied which uses PC to

localize streams of L2 cache misses. SPAD does not use PC and is more suited as an L2 prefetcher. It also outperforms GHB significantly with much smaller hardware cost and complexity.

2.3.5 Variable Length Delta Prefetcher (VLDP)

VLDP [7] maintains a separate local history for each physical page in a Delta History Buffer (DHB), which is a fully-associative table with not-Most Recently Used replacement policy. When a reference to one of these tracked pages results in a prefetching opportunity, the pages delta history is used to look up a prediction in the Delta prediction Table (DPT). The DPT is structured as a set of key-value pairs that associate a delta history within a page with the delta of the next expected access in that page. The VLDP employs multiple DPT tables, where each successive table corresponds to a longer history length. This improves prefetch coverage and accuracy, by preferring to use longer histories to make highly accurate prefetches and using shorter histories to fill in the gaps when long histories are unavailable. VLDP take action only when there are L2 cache misses or when a previously prefetched cache line is accessed.

Similar to AMPM, VLDP requires per-page tracking, however, it does that without keeping each cache line status in the page. Instead, it exploits delta pattern locality and thus requires less hardware. Nevertheless, its fully-associative DHB table, 3 cascaded DPT tables, and multiple actions on them for each cache access make its operation much more complicated than SPAD, which tracks both accuracy and timeliness in a single direct-mapped table. Furthermore, like BOP, VLDP tracks if a cache line is placed into cache due to a prefetch request, which requires one-bit storage per cache line in the cache. SPAD is independent of the size of the cache.

2.4 Motivation

Offset prefetching is a generalization of next-line prefetching. In next-line prefetching, if a cache line A is demand requested, the prefetcher issues a prefetch request for line $A + 1$, i.e., the next sequential line. The issuing of a prefetch can be either on every demand access, or only on cache misses, or on both cache misses and hits on prefetched lines [1]. In offset prefetching, when cache line A is requested, the prefetcher issues a request for cache line $A + o$, where o is a non-zero integer value, the offset. o can be determined statically, i.e. a fixed-offset, or dynamically where a decision mechanism predicts what best offset would be as the program runs.

Recently proposed SBP [1] and BOP [2] (details of which are explained in Section 2.3) show that offset prefetching is relatively simple and outperforms more complicated prefetchers. These are the only two offset prefetchers that we are aware of at the time of this writing. In the following, we analyze offset prefetching and motivate our work.

2.4.1 Performance Potential with Offset Prefetchers

In this section, we analyze fixed-offset sequential prefetching[2] to observe best achieving fixed-offset for each SPEC CPU2006 benchmarks. We simulated sequential prefetching for offsets from -16 to 16 (32 fixed-offsets) for each benchmark. Table 2.1 shows the results for the best performing offset for each SPEC CPU2006 benchmark. Eight benchmarks (416.gamess, 445.gomk, 454.calculix, 458.sjeng, 471.omnetpp, 483. xalancbmk, 433.milc, 453.povray) do not benefit from sequen-

[1]Many prefetchers employ a single bit per cache line to track whether the cache line is placed into the cache due to a prefetch or a demand access. There are multiple reasons. First, one can evaluate the success of prefetching. Second, if prefetches are only issued on cache misses to preserve bandwidth and increase accuracy, prefetching would negatively impact the updates and predictions on prefetchers tables because it changes the miss patterns. Therefore, most prefetchers that issue prefetches on cache misses also issue prefetches when a hit on a previously prefetched cache line occurs.

[2]All the evaluations in this work use L2 level prefetchers

tial prefetching (they are placed under offset 0 (no prefetching) in the table) - the best performing fixed-offset provides less than 0.5% speedup at best. We obtained similar results with other prefetchers that we tested, therefore, we do not consider these seven benchmarks further in this study. Table 2.1 shows that best performing offset varies for each benchmark. Only two benchmarks, 436.cactusADM and 434.zeusmp perform best with offset +1 (i.e., next-line prefetching.) Most benchmarks have their best offsets between +1 and +6. Only 3 benchmarks benefit from negative offsets (and small offsets between -1 and -3).

Since best performing offset varies from benchmark to benchmark, a prefetcher that can automatically find best offset per benchmark would perform the best (hence the adaptivity of recently proposed SBP and BOP). If we are to pick a fixed offset across all benchmarks, we would pick offset 1 as the most commonly observed global and/or local address delta is 1. That is, the resulting prefetcher would be a nextline prefetcher. Figure 2.1 shows, however, that this would be a bad decision. In Figure 2.1, we compare how well various fixed offset prefetchers perform relative to +1 offset prefetcher (i.e., the next-line). We used fixed offsets ranging from 2 to 16. Negative fixed offsets perform very poorly when the same negative offset was used across all SPEC CPU2006 benchmarks. Figure 2.1 shows that offset 1 is clearly not the best fixed offset on the SPEC CPU2006 benchmarks. The best fixed offset is 4. Figure 2.1 also shows the performance when Best Fixed Offset for each Benchmark (BFOB) is used. BFOB performs significantly better suggesting to find methods that exploit this behavior. We also present the results for an oracle offset prefetcher, Oracle, which has prior knowledge of best performing offset for each interval (every 512 L2 accesses) within an application and thus perfectly adapts also to the changing program behavior. Surprisingly, Oracle only marginally (about 0.5%) outperform the best fixed-offset,

Table 2.1: Best Performing Fixed Offset For Each Benchmark.

-3	-2	-1	0	1	2	3	4	5	6	7	10
450	401	473	416	436	465	447	400	429	462	410	481
			445	434	444	456	435	470	459	437	
			454				403	482	464		
			458								
			471								
			483								
			433								
			453								

BFOB, prefetcher. Oracle significantly improve performance relative to BFOB in 4 benchmarks: 450.soplex (4.7% better), 482.sphinx (3.3% better), 401.bzip2 (2.8% better) and 464.h264ref (1.6% better). Nevertheless, the results show that adaptive offset is critical for good performance. Finally, Figure 2.1 also shows the performance results for the recently proposed SBP and BOP[3]. SBP performs better than fixed offset 4 prefetcher. BOP outperforms SBP incorporating timeliness in choosing best offset. There is, however, a significant headroom for improvement. BFOB and Oracle significantly outperform both SBP and BOP.

2.4.2 Understanding the Performance of Offset Prefetchers

To understand where performance really comes from, we try to correlate the most frequently observed global/local deltas within a benchmark with its offset prefetching performance. Table 2.2 shows the most frequently observed global[4] and local (i.e., per PC) deltas for each benchmark. As expected, +1 is the most frequently observed delta globally and locally. 470.lbm and 481.wrf are the only two benchmarks which do not observe delta 1 significantly neither globally nor locally. Although delta 1 is the most frequently observed delta globally and/or locally in

[3]Original BOP uses 46 offsets (23 positive, 23 negative between -40 to +40). In our implementation, SBP performed best with 16 offsets (8 positive, 8 negative, +/-8)

[4]To reduce the interference from interleaved memory accesses, we find global deltas by computing differences (deltas) between current address and the last three access addresses and chose the smallest delta out of the three

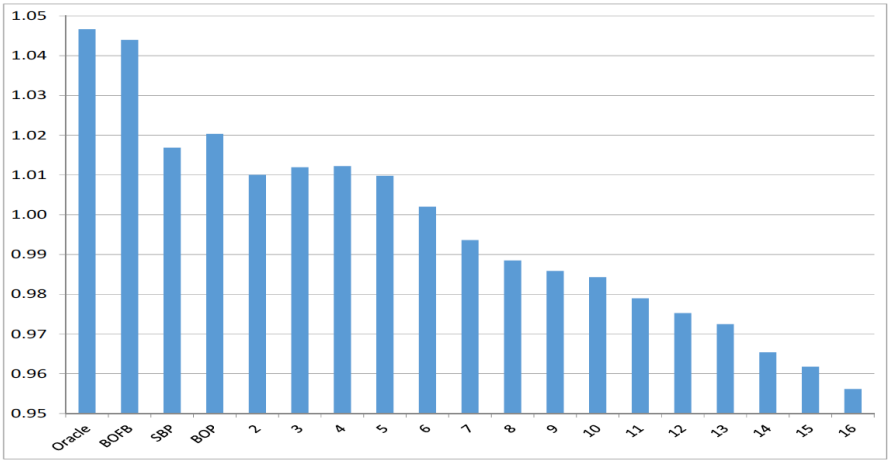


Figure 2.1: Performance with offset prefetching for SPEC CPU2006. x axis shows the Prefetchers: Fixed-offset (2 to 16), Best Fixed Offset per Benchmark (BFOB), Sandbox Prefetcher (SBP) [1] and Best Offset Prefetcher (BOP) [2]. y axis shows the speedup relative to next-line prefetcher (i.e., baseline is the offset 1 prefetcher).

most of the benchmark, only two benchmarks have their best performance with offset 1, as shown in Table I. When we analyze benchmarks individually based on their offset prefetching behavior, we see that for most benchmarks with best performing offset larger than 1, most frequently seen delta is 1 and best performing offset does not appear to be a frequent delta. Most of these benchmarks have significant speedup with offset 1 and the speedup increases as the offset increases peaking at the best offset value. This behavior suggests that offset 1 often issues late prefetches and if prefetch distance increases so does the performance with offset 1 prefetching. That is, offset 1 is the most important offset but it is usually not timely in issuing prefetches. Table 2.3 categorizes the benchmarks based on their offset prefetching behavior. There are 4 categories:

Category 1: 13 out of 21 benchmarks are in this category. Best offset is larger than 1. Most frequently observed global and/or local delta is 1. Offset 1 prefetching provides significant speedup but prefetches are often issued late to fully hide memory latency, therefore a prefetch distance is beneficial. Best offset is equal to delta 1 plus best prefetch distance. Figure 2.2 shows speedup from off-

set prefetching for a number of SPEC CPU2006 benchmarks. 410.bwaves (Figure 2.2c) and 437.leslie3d (Figure 2.2f) are category 1 benchmarks. For 410.bwaves, offset 1 prefetcher (i.e., next-line prefetcher) has a speedup (over no prefetching baseline) of 23%. The speedup increases as offset increases reaching to 43% with offset 7 prefetcher (i.e., the best offset). SBP provides a speedup similar to offset 1 prefetcher. BOP achieves 39%, a little lower than best fixed offset. Result for 437.leaslie3d is very similar: 28% speedup with offset 1 prefetcher, which increases to 41% with best fixed offset prefetcher (BFOB). Again BOP (36%) outperforms SBP (30%) demonstrating that it is successful in integrating timeliness into SBP. However, there is still room for improvement as its performance is lower than BFOB. On average, BOP outperforms SBP on category 1 benchmarks as expected. However, BOP does not always perform well in this category. For example, BOFB provides a 10% speedup over offset 1 prefetcher (i.e., baseline is offset 1 prefetcher) for 456.hmmer with offset 3. BOP only provides 1.2% better than offset 1 prefetcher while SBP outperforms offset 1 prefetcher by 5%. Still about 5% worse than BFOB but 3% better than BOP. Thus, BOP is not able to adapt the behavior of 456.hmmer. 462.libquantum is another benchmark where BOP (also SBP) underperform BFOB significantly.

Almost all benchmarks in this category behave similarly except for 429.mcf. 429.mcf has mostly irregular memory accesses. BFOB (with offset 5) provides only about 6% speedup over no prefetching. This speedup is still due to delta value 1, which is 8% of the global deltas observed. With a prefetch distance of 4, i.e. offset 5, speedup is 3.6% better than next-line prefetching (i.e., offset 1).

Category 2: Best speedup for the benchmarks in this category is provided by offset 1. Most frequently observed delta is also 1. 434.zeusmp and 436.cactusADM are in this category. In this category, prefetch distance hurts the performance

of next-line prefetcher. Figure 2.2d shows that BFOB provides 57% speedup for 434.zeus over no-prefetching baseline. The speedup decreases dramatically as the offset increases. SBP provides 47% speedup while BOP performs poorly with 30% speedup over no-prefetching baseline. BOP is not successful in measuring timeliness for this benchmark. For 436.cactusADM, both BOP and SBP perform as well as next-line prefetcher. Another benchmark in this category is 400.benchmark, which has somewhat interesting behavior (possibly that could fit in another category). The best speedup is obtained with offset 4, 27%. Speedup with offset 1 is 21%. Speedups with offsets 2 and 3 are not better than offset 1. A delta value other than 1 provides additional speedup. Table 2.2 shows that second most observed local delta is 2. This explains why offset 4 prefetcher provides the best speedup. Because of its behavior, 400.perlbench is also placed in category 3.

Category 4: This category consists of benchmarks that have the best speedup with negative offsets. Only 3 benchmarks have significant performance gains with negative offsets. The most frequently observed negative delta is -1 in all three benchmarks. 473.astar only benefits from offset -1. offset -2 reduces the speedup significantly. Offset -3 does not provide any speedup. For 450.soplex, offset -3 is the best performing offset. However, offset -1 provides significant speedup and increasing negative offset further increases the performance. With their offset prefetching behavior, 473.astar is similar to the benchmarks in category 2 and 450.soplex is similar to the behavior of benchmarks in category 1, but with negative deltas. Finally, 401.bzip2 performs best with offset -2 achieving a 7.5% speedup over no prefetching baseline. However, its behavior is rather unusual (see Figure 2.2a). For offset -1, there is no speedup but a slight slowdown. And for offsets smaller than -2, speedup drops to less than 2%. Considering that the most frequently observed delta is -1, this benchmarks behavior is hard to capture for timeliness of prefetches.

Table 2.2: Frequently Observed Global and Local Deltas

	global				**local**			
benchmarks	**delta 1**		**delta 2**		**delta 1**		**delta 2**	
400.perlbench	1	29%	64	2%	1	44%	2	14%
401.bzip2	1	28%	-1	21%	1	29%	-1	23%
403.gcc	1	49%	-1	3%	1	43%	2	7%
410.bwaves	1	50%	64	24%	1	93%	2	3%
429.mcf	1	8%	24	4%	88	14%	24	8%
434.zeusmp	1	16%	16	16%	1	36%	17	23%
435.gromacs	1	30%	64	6%	1	15%	3	14%
436.cactusADM	15	7%	-30	6%	1	95%	2	2%
437.leslie3d	1	35%	64	9%	1	94%	2	2%
444.namd	1	61%	62	3%	1	54%	2	14%
447.dealII	1	75%	-1	10%	1	76%	-1	9%
450.soplex	1	43%	-1	8%	1	28%	-1	25%
456.hmmer	19	20%	64	9%	1	74%	2	8%
459.GemsFDTD	1	28%	29	8%	1	76%	29	12%
462.libquantum	1	97%	-3	2%	1	95%	6	2%
464.h264ref	1	52%	2	2%	1	61%	81	4%
465.tonto	1	51%	2	3%	1	57%	2	3%
470.lbm	-4	6%	-52	5%	2	37%	3	37%
473.astar	1	15%	-1	11%	1	9%	64	8%
481.wrf	10	17%	2	10%	10	43%	9	14%
482.sphinx3	1	52%	64	2%	1	66%	2	4%

SBP does considerably well achieving a 3.9% speedup while BOP performs worse with a 1.8% speedup.

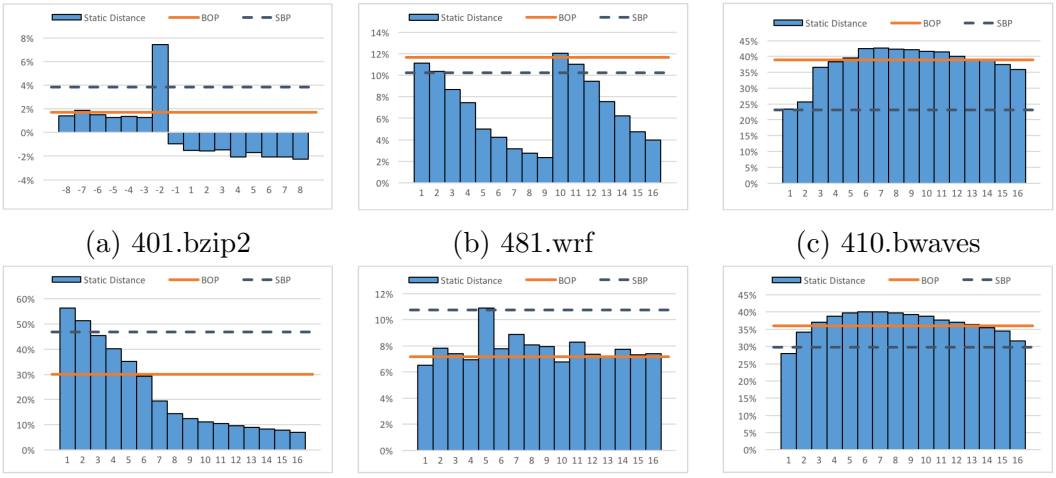
2.4.3 Our Motivation

We can draw important conclusions from the analysis of offset prefetching:

1. Best offset varies for each benchmark. Most best offsets are small, 1 to 6. BOP and SBP are recent proposals that aim to find the best offset. However, although BOP and SBP perform better than recent prefetchers that exploit regular memory access patterns, there is a significant room for improvement. BFOB outperforms BOP by 3% as shown in Figure 2.1.

Table 2.3: Benchmark Categories Based on Offset Prefetching Behavior

Category 1	Category 2	Category 3	Category 4
403, 410, 429, 435, 437, 444, 447, 456, 459, 462, 464, 465, 482	434, 436, 400	470, 481, 400	450, 473, 401



(a) 401.bzip2 (b) 481.wrf (c) 410.bwaves

(d) 434.zeusmp (e) 470.lbm (f) 437.leslie3d

Figure 2.2: Performance comparison of sequential prefetching with static offsets, BOP, SBP on some benchmarks.

2. For most benchmarks (19 out of 21), the performance benefit comes from exploiting global and/or local delta value 1 or -1 (negative delta is only important for category 4, that is, for 3 benchmarks). However, timeliness is important. Only two benchmarks in category 3 are the outliers.

3. Category 1 and 2 benchmarks have regular behavior that can simply be exploited with a next-line prefetcher with adaptive distance.

Based on our findings, we propose a simple yet effective sequential prefetching mechanism with adaptive distance, SPAD. Since most performance benefit in offset prefetching comes from delta value 1 but improved prefetch timeliness than next-line prefetcher (i.e., offset 1 prefetcher), SPAD employs a delta 1 prediction with

a feedback mechanism to predict best prefetch timing for delta 1. SPAD tracks issued prefetches using a table called Testing Queue (TQ) to guide their timeliness by adjusting the prefetching distance. In the next section, we describe the details of SPAD.

2.5 SEQUENTIAL PREFETCHER WITH ADAPTIVE DISTANCE (SPAD)

Figure 2.3 illustrates our proposed SPAD prefetcher. *Distance* represents the estimated distance, in cache lines, that would be added to the current demand-requested cache line address to generate the prediction for a prefetch address. Since SPAD does not prefetch across page boundaries, maximum *Distance* cannot be greater than 63 assuming 4KB pages and 64B cache lines. *Distance* 1 represents a next-line (i.e., offset 1) prefetcher, *Distance* 2 represents a next-line prefetcher with a prefetch distance of 1 and so on. *Distance* 0 represents no prefetching case (i.e., prefetching turned off). When a read request for line A accesses the L2 cache, if A and $A + Distance$ lie in the same physical page, a prefetch request for $A + Distance$ is sent to lower-level memory. SPAD increments or decrements the Distance after an evaluation period, trying to adapt to the application behavior by finding the best prefetch distance for delta 1. If prefetching is determined to be harmful, SPAD turns off the prefetching by resetting the *Distance* to zero.

2.5.1 Test Queue (TQ)

In order to estimate prefetch timeliness with current Distance, SPAD records predicted prefetch addresses (shown as test address in Figure 2.3) in a table called the Test Queue (TQ). Several implementations are possible for the TQ. In this study, we choose the simplest implementation: TQ is direct-mapped, accessed through a simple index function, each entry holding a tag. The tag does not need to be the full address, a partial tag is sufficient. In our implementation, for a

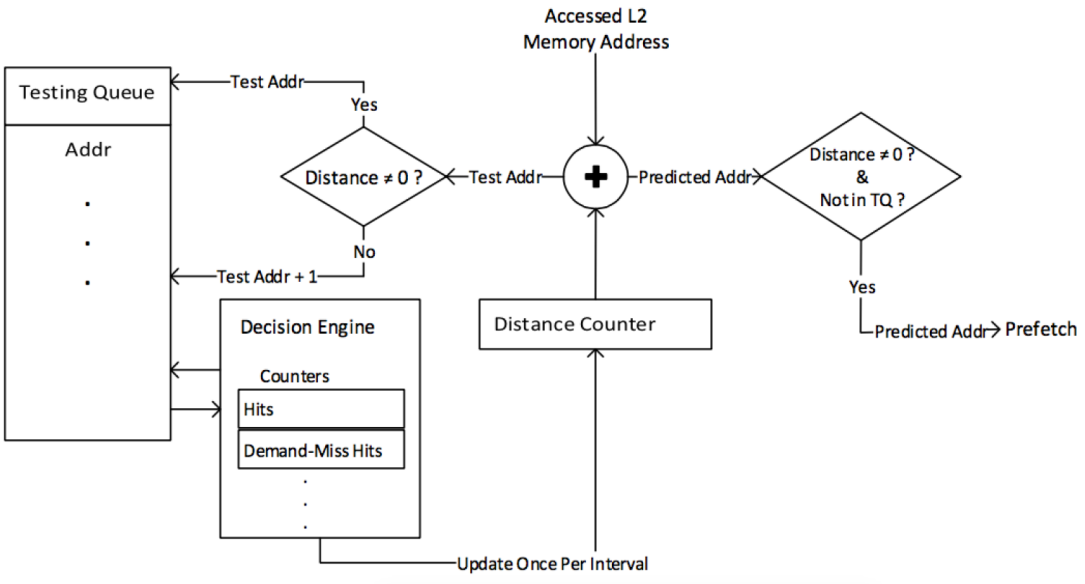


Figure 2.3: The Proposed SPAD prefetcher components.

128-entry TQ, we use the 7 least significant line address bits to index the table. For 8-bit tags, we skip the 7 least significant line address bits and extract the next 8 bits.

Each L2 cache demand access (e.g, to line A) triggers a prediction for a candidate prefetch address. After the line address, $A + Distance$ is predicted as a prefetch candidate, if it is not already in the TQ, it is recorded in the TQ and a prefetch for $A + Distance$ is issued. If it is found in the TQ, no prefetch is issued. That is, the TQ also acts as a prefetch filter in order to filter redundant prefetch requests going to the memory system.

When Distance is zero, no prefetching would occur. However, the TQ continues to record $A + 1$ (next-line) as if an offset 1 prefetcher is active. This is needed to continue evaluating the prefetcher when it is off so that later when the prefetching is useful again, it can get reactivated.

2.5.2 Interval

Distance is updated at the end of each evaluation period called an Interval. In this work, an interval is a period of 512 L2 cache accesses. At the end of each interval, SPADs Decision Engine (DE) makes a decision to update the Distance (increment, decrement, or zero), if needed. This decision is made based on the counter values that track a number of events (e.g., L2 misses) in the last interval as described in the following section (Section 2.5.3). After each interval, all counters are reset to zero and a new evaluation period (i.e., a new interval) starts. For each L2 demand access in an interval, SPAD predicts prefetch addresses using the last *Distance* value set by the DE. If *Distance* is 0, no prefetch is issued during that interval but next-line prefetcher (i.e., $Distance = 1$) continues to be evaluated.

2.5.3 Decision Engine (DE)

After each interval, DE updates the *Distance* as necessary based on three counters: $l2miss$, $tqhits$ and $tqmhits$. $l2miss$ tracks the number of total L2 cache misses in an interval. $tqhits$ is the number of L2 demand accesses that are found in the TQ in an interval, and $tqmhits$ tracks the number of L2 demand misses that are found in the TQ in an interval. After each interval, DE checks several conditions to make distance update decision as follows.

1. If $tqhits < TQTHLD$ (64 in our evaluation) and if $Distance > 1$, *Distance* is decremented. The intuition behind this action is that $tqhits$ is low because either predicted addresses are not accurate or prefetches are issued too early so that they are not in TQ anymore (replaced by other predictions).

2. If $tqhits < TQTHLD$ for three consecutive intervals, prefetching is considered useless and *Distance* is reset to zero disabling the prefetching.

3. If $tqhits >= TQTHLD$, update decision for *Distance* is made as follows (in

this order):

(a) If $l2miss < MISSTHLD$ (8 in our evaluation), no update is made assuming current *Distance* is successful.

(b) Finally, if $tqmhits > l2miss/2$ for more than two consecutive intervals, distance is incremented. No division is necessary for this check, a simple shift operation is sufficient. The intuition behind this decision is that when most L2 misses are found in the TQ, although prediction accuracy is high, prefetches are likely issued too late.

4. Since prefetcher continues to record predicted addresses (i.e., $A + 1$) in the TQ when prefetching is off (i.e., *Distance* is zero), prefetching can be turned back on if it is proved successful. In our implementation, this is measured by the following condition: $tqhits >= TQTHLD$ for two consecutive intervals.

2.5.4 Integrating Negative Distance Prediction into SPAD

For a few benchmarks, negative deltas are most important for performance. These benchmarks are placed under Category 4, as described in Section 2.4.2. We can easily integrate negative *Distance* prediction into SPAD without needing significant extra hardware. Similar to SBP and BOP, where many offsets are tested using a single table, SPAD can use the same TQ to evaluate *Distance* for both delta 1 and -1, each taking a turn for one interval. That is, after delta 1 with *Distance* is evaluated for an interval, all counters, except for Distance and interval tracking counters, are reset to zero and delta -1 with *Distance* is evaluated for an interval in a round-robin fashion. So, there are two prefetchers, one for delta 1 and one for delta -1. This requires to have separate Distance counters, also separate counters for tracking consecutive intervals for each. Overall, this adds up only 12 bits (6 bits for *Distance* counter and 6 bits for three 2-bit counters tracking

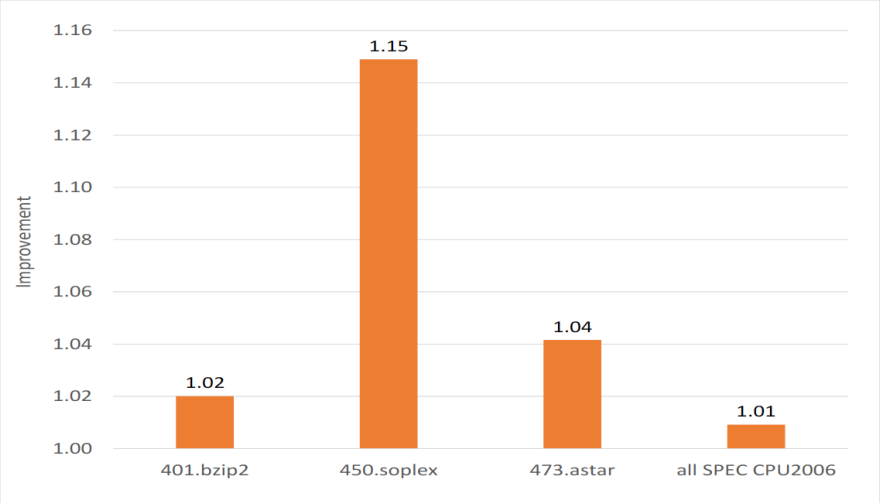


Figure 2.4: Performance improvement when negative *Distance* prediction is integrated into SPAD. Baseline is SPAD with no negative *Distance* prediction

consecutive intervals (see Section 2.5.3)) to the delta 1-only case.

In our implementation, both delta 1 and delta -1 prefetcher is active in SPAD if their *Distance*s are not zero. In most benchmarks, though, we observe that prefetching for delta -1 is off most of the time. Overall, adding negative *Distance* predictions to SPAD improve its performance on SPEC CPU2006 benchmarks by about 1%, as shown in Figure 2.4. This additional performance comes from the Category 4 benchmarks (450.soplex, 473.astar and 401.bzip2). Most significant improvement is for 450.soplex, with about 15% IPC improvement over SPAD without negative delta integration. The performance impact on Category 1, 2 and 3 benchmarks is insignificant and therefore is not shown in Figure 2.4.

2.6 Methodology

2.6.1 Simulation Environment

We implemented SPAD in the 2015 Second Data Prefetching Competition (DPC-2) Framework [3] provided by Intel. We use the competitions baseline hardware configuration because BOP (winner of the DPC-2), VLDP and AMPM were optimized for this configuration. For BOP, VLDP and AMPM (our results show

Table 2.4: Simulator Parameters [3]

Processor	
Core Parameters	Intel x86 ISA Core Parameters 256-entry instruction window 6-wide out-of-order core
Cache Hierarchy	
L1 I & D Caches	16KB, 8-way, LRU, 1-cycle
L2 Cache	128KB,10-cycle L2 Cache 64B, 8-way LRU, shared Request Queue 32, MSHRs 16
L3 Cache (LLC)	1MB,30-cycle64B, 16-way LRU, shared
DRAM	
Frequency	1600 MT/s (12.8GB/s)
Channel, rank, bank	64-bit channel, 1 rank, 8 banks
DRAM core access latency	row hit: approx. 13.5ns row miss: approx. 40.5ns

Slim AMPM [26], an improved version submitted to DPC-2), we used their source code available at the DPC-2 website. Hence, these prefetchers are authors optimized versions for the DPC-2 framework. This is one of the reasons why we have used DPC-2 framework in our evaluation. Our SPAD implementation optimized for DPC-2 framework allows a fair comparison.

2.6.2 Simulator Parameters

The DPC2 framework models a 6-wide issue out-of-order core with parameters summarized in Table 2.4. All prefetchers evaluated in this work are at the L2 cache level. We run all our experiments with a shared 128KB L2 cache because BOP, VLDP, and AMPM were optimized for this configuration in the DPC-2 competition.

Table 2.5: SPAD Prefetcher Default Parameters

TQ table entries	128
TQ tag bits	8
TQTHLD	64
TQTHLD	32
TQTHLD	512

2.6.3 SPAD Hardware Budget

Table 2.5 provides the default parameters of the SPAD prefetcher evaluated in this work. The major storage needed for SPAD is for its TQ. In our evaluation, we use a 128-entry TQ. Each entry stores 8 bits of line address tag. SPAD employs five 9-bits counters (since interval size is 512), $l2miss$, $tqhits$, $tqmhits$, and $l2acc$, and 6 2-bit counters (for tracking number of consecutive intervals a condition repeats) to keep track of several events as described in Section 2.5.3. In addition, SPAD needs a 9-bit interval register, two 6-bit *Distance* registers, one for delta 1 and one for delta -1, one 8-bit register to hold $TQTHLD$ and one 4-bit register for $MISSTHLD$. Overall hardware budget is 1105 bits or 139 bytes.

2.6.4 Benchmarks

In our evaluations we used SPEC CPU2006 [27] benchmark suite. We used Simpoint 2.0 [28] to generate representative 100M-instruction traces. The measurements in the early part of the cycle-accurate simulations are discarded to account for various warm-up effects.

2.7 Results

We simulate SPAD with a 128-entry direct-mapped TQ table and an interval size of 512. We compare our results to five state-of-the-art prefetchers that exploit regular memory access patterns. Section 2.3 contains their descriptions. SPAD also focuses on regular patterns. Our work is most related to recently proposed offset prefetchers SBP [1] and BOP [2], which outperform prior work. Table 2.6 details

Table 2.6: Prefetcher Storage Overheads and Parameters

Prefetcher	**Storage**	**Parameters**
SBP	296B	256B Bloom filter, 16 candidates
BOP	545B	RR table (2x64 entries, 12-bit tags), Delay queue (15x31 bits, 2 pointers)
AMPM	2KB	64 Access Maps, each tracking 4KB
GHB	4KB	1024 IT entries, 1024 GHB entries
VLDP	1496B	OPT 392B, DHB 552B, DPT 552B
SPAD	139B	TQ: 8-bit entry, 128 entries, interval size: 512

the storage requirements and basic parameters for SPAD and competing schemes. This table shows that SPAD uses much smaller hardware than the competitor prefetchers. Its hardware budget is only one-fourth of the BOPs, which is the best performing competitor, and half of the SBPs.

In the following, we first explore SPADs design space to find the best performing interval length and TQ size. We then present the performance evaluation of SPAD and comparison with prior methods.

2.7.1 Finding Best SPAD Parameters

SPAD has 4 important parameters: interval length, TQ size, $TQTHLD$, and $MISSTHLD$. Interval length is the most important parameter because it determines the other parameter values. In our experiments, we observe that best threshold values are $TQTHLD = intervallength/8$ and $MISSTHLD = intervallength/16$. For example, for an interval length of 512, the best performing $TQTHLD$ is 64 and $MISSTHLD$ is 32.

Interval Length: In order to evaluate the effect of interval size on SPADs performance, we kept TQ size constant at 128, while varying the interval length. Threshold values change with the interval length.

Figure 2.5 shows the geometric mean speedups for all SPEC CPU2006 benchmarks with SPAD for varying interval lengths. We can see that SPAD has signifi-

cantly lower performance when interval length is smaller than the TQ size. Smaller interval lengths do not provide a sufficient number of accesses to monitor and make decisions accurately. Interval sizes longer than 1024 also cause performance loss, but only slightly. As the interval length gets larger, it may get too large for SPAD to adjust the offset on time. If the program behavior changes or the current offset is not the best performing one for some reason, it should be updated quickly to recover lost prefetching opportunities. An interesting observation is that the performance is still relatively good for very large interval length. This suggests that most predictions recorded in the TQ were accessed within 128 L2 accesses and thus, SPAD continues to make correct decisions.

Figure 2.5 shows that the best performance is obtained when the interval length is 512, or 4 times the TQ size. One might think that interval length must be less than or equal to the number of TQ entries to possibly record all predictions in the TQ within an interval. However, keeping distant predictions in the TQ would inflate the number of tqhits, which might, in turn, make it hard to evaluate timeliness and prefetchers performance. Therefore, we conclude that the interval length must be larger than TQ size.

Effect of TQ Size: To study the impact of TQ size on performance, we simulated SPAD with an interval length 4 times the size of the TQ and vary the number of TQ entries from 32 to 256. Figure 2.6 shows the normalized speedups for each SPEC CPU 2006 benchmarks with varying TQ size using 32-entry TQ as our baseline. Interval sizes and thresholds for each TQ size is given in Table 2.7. Our results show that 128-entry TQ performs best for most of the benchmarks. 64-entry TQ seems to be significantly beneficial for 436.cactusADM. A 256-entry TQ works better for 437.leslie3d. 32-entry TQ is slightly better for 482.sphinx. On average, 128-entry TQ performs best, however. Therefore, the best configuration

Table 2.7: Parameters used for evaluating the effect of TQ size.

Number of TQ Entries	32	64	128	256
Interval Length	128	256	512	1024
TQTHLD	16	32	64	128
MISSTHLD	8	16	32	64

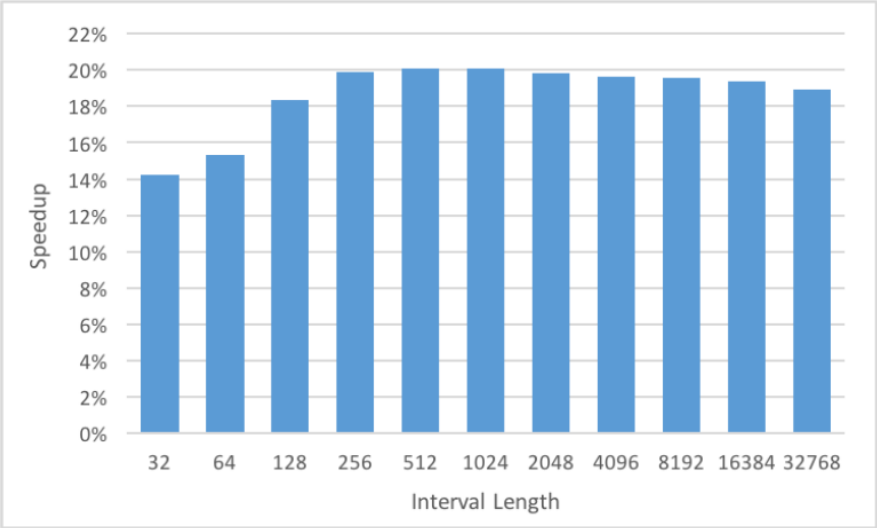


Figure 2.5: Impact of interval length on performance. Figure shows the geometric mean speedups of SPAD for SPEC CPU2006 benchmarks with varying interval lengths. TQ size is 128 entries.

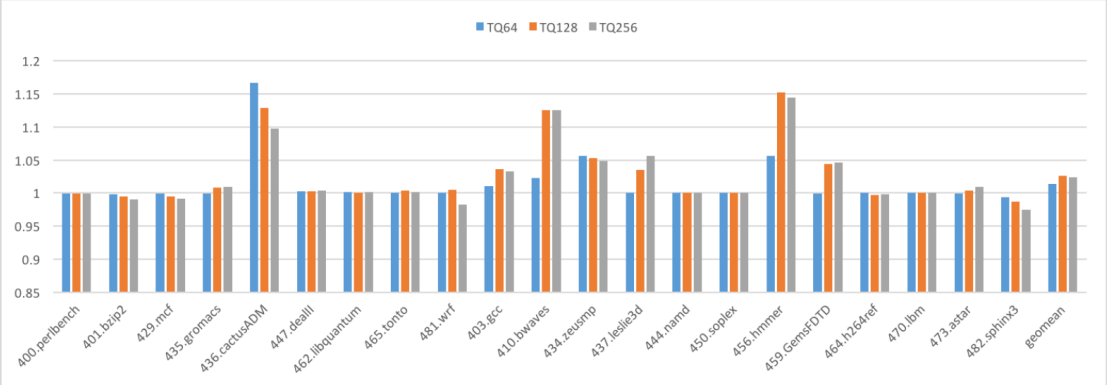


Figure 2.6: Effect of Varying TQ size. Figure shows the IPC results for varying TQ sizes normalized to IPC for the case where TQ size is 32. Interval size is 4 times the TQ size.

for SPAD is using 128-entry TQ and an interval size of 512. In the remainder of the chapter, our experiments use this configuration.

2.7.2 Performance Evaluation

Figure 2.7 shows the speedup of SPAD in comparison to SBP, BOP, VLDP, GHB, and AMPM. The results in the figure are presented separately for each benchmark category. For Category 1 benchmarks, delta +1 is most important for performance but they require a prefetch distance for timely prefetches. Most SPEC CPU2006 benchmarks fall into this category. SBP does not consider prefetch timeliness in its offset evaluation and thus perform worse than BOP and SPAD. SBP performs very poorly for 410.bwaves. While SPAD and BOP achieve 42% and 38% speedups for this benchmark, respectively, SBP only achieves a 23% speedup. For 410.bwaves, the best fixed offset is 7. SBP finds best common offset +1 for this benchmark and performs only as well as a next-line prefetcher. SPAD and BOP can incorporate prefetch timeliness well for this benchmark and hence, perform well. SPAD outperforms SBP on all benchmarks in Category 1. On average, SPAD performs the best in this category followed by BOP. SPAD outperforms BOP on 7 benchmarks, while BOP outperforms SPAD on 3 benchmarks in this category. AMPM is the third best, followed by SBP, VLDP, and GHB.

As for Category 1, most important delta for performance is +1 in Category 2. Unlike Category 1, however, prefetching without prefetching distance (i.e., next-line prefetching) perform best for Category 2 benchmarks. For 434.zeusmp, SPAD performs significantly better than both SBP and BOP. BOP fails to score the best performing offset, which is +1, for this benchmark, and thus performs very poorly. Surprisingly, SBP also does not do as well as SPAD (or a next-line prefetcher) for 434.zeusmp because best scoring offset may not always be the best performing offset. For 436.cactusADM, SPAD, SBP and BOP, all perform well. 400.perlbench has two different deltas significant for performance as discussed in Section 2.4.2. That is why it is placed in Category 3 as well as in this category. SPAD focuses on

delta 1 prediction with adaptive distance therefore only capture delta 1 behavior. SPAD performs as well as an offset +1 prefetcher on 400.perlbench while SBP is able to find both important offsets for this benchmark and perform better than BOP and SPAD. VLDP performs best for 400.perlbench exploiting the multi-delta sequences in this benchmark. On average, SPAD outperforms all prefetchers in Category 2. SBP is a close second followed by BOP.

Deltas other than 1 have a significant performance impact for Category 3 benchmarks. 481.wrf benefits from delta 10 prefetching. For 470.lbm, 2, 3 and 5 are the most important deltas. Best fixed offsets for 481.wrf and 470.lbm are 10 and 5, respectively. SPAD is designed to capture the timeliness of delta 1 prefetching. It cannot exploit the type of behavior Category 3 benchmarks exhibit. SBP performs best in this category. VLDP performs relatively well compared to its performance for Category 1 and 2. It is second best in Category 3.

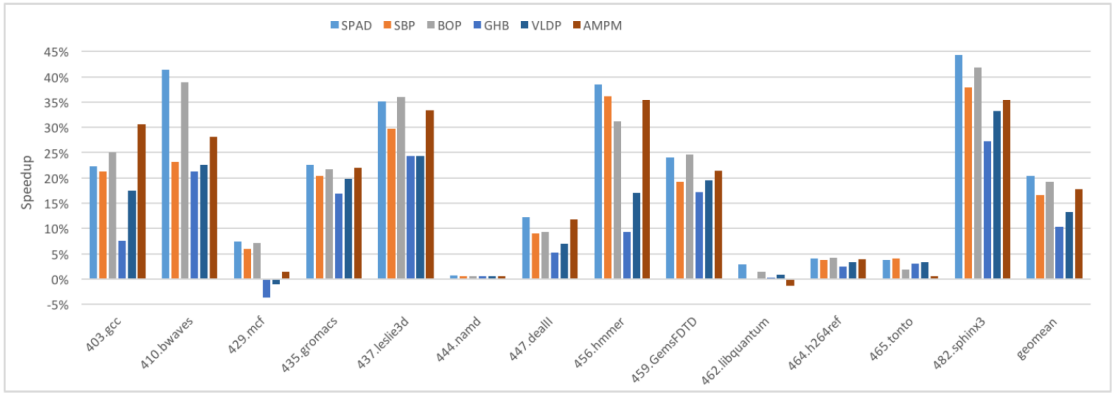
Finally, in Category 4, negative deltas have a significant impact on performance. Only 3 benchmarks exhibit this behavior. BOP performs worse on 401.bzip2 while it performs best on 450.soplex. However, the performance differences for benchmarks in this category are very small. SBP performs best for 401.bzip2 and 473.astar. All prefetchers, except for GHB, perform similarly in this category. GHB has significantly lower performance in all categories.

Figure 2.8 summarizes the overall performance results. SPAD achieves 20.1% speedup over no-prefetching baseline, while BOP and SBP achieve 18.8% and 18.4% speedup, respectively. AMPM provides 17% speedup. VLDP and GHB perform significantly lower with 14% and 11% speedup, respectively. SPAD performs better than all the competitor prefetchers and with a lower hardware budget. SPAD outperforms BOP by 1.1% with one-fourth of the hardware budget. It outperforms SBP by 1.5% with less than half the hardware budget. VLDP, AMPM,

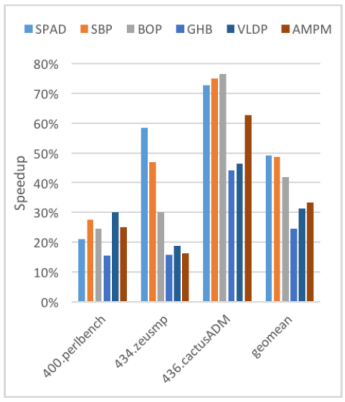
and GHB use significantly more hardware and perform significantly worse than SPAD. Figure 2.8 also provides the speedup for an ideal offset prefetcher, called the ORACLE. In order to implement the ORACLE offset prefetcher, we first determine best performing offset for each interval within a benchmark through extensive simulations. The pre-extracted offset list is used by ORACLE as input and each offset in the list is used for prefetching at their corresponding interval. The ORACLE prefetcher achieves 22% speedup over no prefetching baseline. SPAD matches the speedup of all Category 1 benchmarks except for 482.sphinx. SPAD also performs as well as the ORACLE for Category 2 benchmarks, with the exception of 400.perlbench (which is in category 2 partially). This shows that SPAD issues very timely prefetches for the categories of benchmarks that it is designed for. For Category 3 and 4 benchmarks, ORACLE significantly outperforms SPAD, as expected. Most notable is 401.bzip2. ORACLE outperforms SPAD by 10% for 401.bzip2. On average, ORACLE performs only 1.5% better relative to SPAD.

2.8 Conclusion

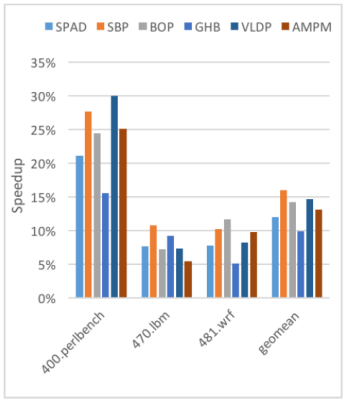
Variants of next-line sequential prefetchers have commonly been employed in current processors due to their good performance and simplicity. In next-line prefetching, when a line A is demand accessed, a prefetch is issued for line $A + 1$. Recently proposed offset prefetching and the sandbox technique [1] generalizes next-line prefetching by sending a request for line $A + O$, where O is a non-zero offset value when line A is demand accessed. Pugsley et al. [1] has shown that their Sandbox Prefetcher (SBP) outperforms previous prefetchers that target regular memory access patterns. SBP, however, does not consider prefetch timeliness. Best Offset Prefetcher (BOP) [2] proposed a solution where prefetch timeliness was integrated into SBP-like prefetcher. Just like SBP, BOP tests many different offsets by recording their line predictions in a table and scoring their accuracy by



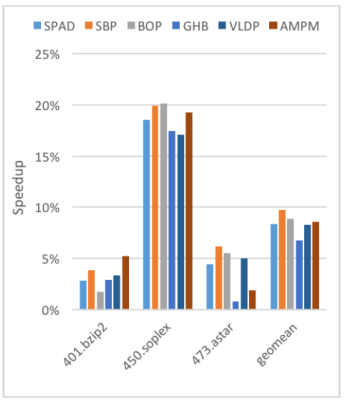
(a) Speedup for Category 1 benchmarks



(b) Speedup for Category 2 benchmarks



(c) Speedup for Category 3 benchmarks



(d) Speedup for Category 4 benchmarks

Figure 2.7: Comparing the Performance of SPAD to SBP, BOP, VLDP, GHB and AMPM.

counting the number of demand access hits in the table. However, the scoring of offsets also takes into account prefetch timeliness. BOP, the winner of 2015 Data Prefetching Championship (DPC-2), has shown that it outperforms SBP by finding best offsets more successfully.

In this chapter, we analyzed offset prefetching and realized that benchmarks often perform best with offsets larger than 1 but they exhibit regular delta 1 memory access patterns. What makes a specific offset work best is not because memory access sequences exhibit a delta value that is equal to the offset, but because that offset value provides a prefetch distance for the delta value 1 performing

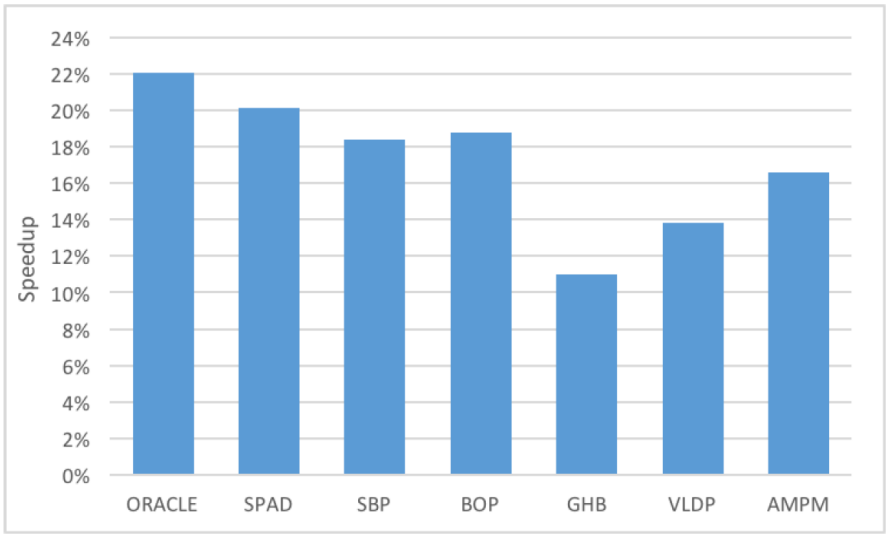


Figure 2.8: Overall Performance Comparison of SPAD, SBP, BOP, GHB, VLDP, AMPM and ORACLE. Figure shows geometric mean speedup for all SPEC CPU2006 benchmarks. Oracle shows the best possible speedup by offset prefetching.

the prefetch in a more timely manner. We proposed the Sequential Prefetcher with Adaptive Distance (SPAD) to exploit this behavior. SPAD focuses on delta value 1 and the prefetch timeliness. Instead of testing many offset prefetchers, it tests only one prefetcher (delta value 1) and tracks best prefetch distance for timeliness. Our results show that SPAD outperforms SBP and BOP by 1.5% and 1.1%, respectively, with a simpler mechanism and much lower hardware budget.

List of References

[1] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 626–637.

[2] P. Michaud, "A best-offset prefetcher," in *2nd Data Prefetching Championship*, 2015.

[3] "Dpc-2. second data prefetching championship." 2015. [Online]. Available: http://comparch-conf.gatech.edu/dpc2/

[4] M. Cavus, I. B. Karsli, and R. Sendag, "Array tracking prefetcher for indirect accesses," 2015.

[5] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 499–500.

[6] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 96–96.

[7] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 141–152.

[8] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[9] K. So and R. N. Rechtschaffen, "Cache operations by mru change," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.

[10] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

[11] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

[12] I. Sklenář, "Prefetch unit for vector operations on scalar computers," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 4, pp. 31–37, 1992.

[13] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 364–373.

[14] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

[15] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *1993 International Conference on Parallel Processing-ICPP'93*, vol. 1. IEEE, 1993, pp. 56–63.

[16] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 397–408.

[17] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 63–74.

[18] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 252–263.

[19] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004, pp. 135–145.

[20] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 252–263.

[21] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 301–313.

[22] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 69–80.

[23] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, no. 2011, pp. 1–24, 2011.

[24] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 247–259.

[25] "Dpc-1.1stjilpdataprefetchingchampionship." 2009. [Online]. Available: http://www.jilp.org/dpc/

[26] V. Young and A. Krishna, "Towards bandwidth-efficient prefetching with slim ampm," *The 2nd Data Prefetching Championship*, 2015.

[27] "Standard performance evaluation corporation cpu2006 benchmark suite." [Online]. Available: http://www.spec.org/cpu2006/

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

CHAPTER 3

Informed Prefetching for Indirect Memory Accesses

Mustafa Cavus[1], Resit Sendag[2]

[1,2]Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI02882.

3.1 Abstract

Indirect memory accesses have irregular access patterns which limit the performance of conventional software and hardware-based prefetchers. To address this problem, we propose the Array Tracking Prefetcher (ATP) [1] which tracks array-based indirect memory accesses using a novel combination of software and hardware. ATP yields an average speedup of 2.17 as compared to a single-core without prefetching. By contrast, the speedup for conventional software and hardware-based prefetching is 1.84 and 1.32, respectively. For four-cores, the average speedup for ATP is 1.85, while the corresponding speedups for software and hardware-based prefetching are 1.60 and 1.25, respectively.

3.2 Introduction

Traversing sparse matrices and graphs frequently results in indirect memory accesses, which have irregular access patterns and thus poor cache spatial locality. These data structures are often implemented as nested arrays, e.g., $A[B[i]]$, wherein the index of the outer array is the value stored in a memory location within the inner array. A hardware stream prefetcher can effectively prefetch entries of the array B because its entries are accessed sequentially, thus having good spatial locality. By contrast, because there may be no pattern to the values stored in the array B, there is likewise no pattern in the accesses to array A, which diminishes the efficacy of a hardware stream prefetcher.

Software prefetching can hide the memory latencies of indirect memory accesses, but requires executing additional instructions (e.g., the prefetching instructions themselves, instructions for address calculation and border checking, etc.). See, e.g., [2] (describing a compiler-based system to generate software prefetches for indirect memory accesses). Figure 3.1 shows the percentage increase in the number of instructions in a loop iteration due to software prefetching for various

benchmarks. For some benchmarks, e.g., *Integer Sort (is)*, and *Conjugate Gradient (cg)*, the overhead is very high (e.g., approximately 100%) as those benchmarks have relatively few instructions in each iteration. *Graph500 (g500)* has a high instruction overhead due to border checking instructions in the frequently executed loop. While the benefit of prefetching may offset the overhead for some benchmarks (e.g., is and *g500*), for others (e.g., *cg*, *Hashjoin ph 2 (hj2)*, and *Hashjoin ph 8 (hj8))*, the reverse is true.

In addition to instruction overhead, the benefit of software prefetching is further limited by (1) dependences related to prefetch address calculation, which may reduce the prefetch distance (i.e., how far in advance of the memory access the prefetch instruction is issued), and (2) the lack of run-time information, which is required to optimally place the prefetch instructions. Figure 3.2 depicts the effect of prefetch distance on speedup for two benchmarks, Histogram (histo) and PageRank (pr). For histo, the highest speedup occurs at the largest prefetch distance (128) while the speedup is lower for smaller distances, especially for prefetch distances of 1, 2, 4, and 8. The opposite is true for pr, namely, the highest speedups occur at the smallest prefetch distances. For a set of memory-bound benchmarks with indirect memory accesses, the prefetch distance has a very significant effect on the speedup of software prefetching. More specifically, the average speedup across all benchmarks ranges from 1.19 (worst) to 1.84 (best). Finally, it is also important to remember that the optimal prefetch distance for a given application may change based on running the application on a different underlying architecture [2], which further underscores the necessity of run-time information to optimally place the prefetch instructions.

Hardware prefetchers, by contrast, do not require executing additional instructions in order to compute and issue prefetches. But, in order to capture irregular

Code Snippet 3.1: Two-dimensional array in a nested loop.

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    load A[B[i][j]]
```

Code Snippet 3.2:) Index requires arithmetic/logical computations.

```
for (i = 0; i < N; i++)
  load A[(B[i]&0x3f)>>2]
```

access patterns, hardware prefetchers generally require very complex mechanisms. See, e.g., Hashemi et al. [3] (describing continuous runahead execution). Yu et al. [4] proposed a pure hardware mechanism called Indirect Memory Prefetcher (IMP) which was designed to capture a few different indirect memory access patterns (e.g., $A[B[i]]$ and $A[B[C[i]]]$).

Code Snippets 3.1-3.3 below illustrate the limitations of hardware prefetching and concomitantly the advantages of software prefetching. We use IMP as an exemplary hardware prefetcher given its efficacy and relatively low complexity. Code Snippet 3.1 depicts indirect memory access where the index array, i.e. B, is a multidimensional array. This type of code appears in benchmarks such as *PageRank (PR)* and *Triangle Count (TC)*. A hardware prefetcher like IMP can capture this indirect memory access by prefetching $A[B[i][j + D]]$ where D is the prefetch distance. Even when the maximum number of iterations for the inner loop is very small, IMP can still capture these indirect memory accesses, but it may not be able to fully hide the memory latency as the prefetch distance is too small. But, in this case, increasing the distance actually decreases performance because the inner loop is too short. Software prefetchers solve this problem by prefetching memory accesses for the next iteration in the outer loop, e.g., $A[B[i + 4][j]]$; hardware prefetchers, however, have trouble detecting this behavior.

Code Snippet 3.3: Code requires simultaneous tracking of multiple indirect accesses of varying depth using the same index array.

```
for (i = 0; i < N; i++)
  load A[B[C[i]]]
  load D[C[i]]
```

Code Snippet 3.2 depicts indirect memory access that requires arithmetic/-logical operations to compute the memory address. More specifically, the index to array A requires a logical AND and a right-shift. This type of code appears in benchmarks such as *HashJoin ph2 (hj2)*. Most hardware prefetchers such as IMP cannot capture these memory accesses because they require more than one arithmetic/logical operations; expensive hardware prefetchers, e.g., continuous runahead execution [3], can successfully prefetch this type of indirect memory access but only when runahead is sufficiently far and dependence chain can be successfully detected.

Code Snippet 3.3 depicts an example of when multiple indirect accesses of varying depth appear at the same time. In this situation, because tracking these memory accesses in hardware is very complicated, IMP does not capture the full behavior. More specifically, IMP was able to detect and prefetch $B[C[i]]$ and $D[C[i]]$, but not $A[B[C[i]]]$. IMP can track $A[B[C[i]]]$, but only if it does not exist at the same time as $D[C[i]]$.

By contrast, software prefetching can accurately prefetch the memory accesses depicted in the above code snippets, but only with substantial programmer effort. Also, prefetching these memory accesses requires significant overhead because it requires performing the arithmetic/logical operations for every prefetch. Lastly, the best prefetching distance is hard to predict due to the lack of run-time information.

Given that software and hardware prefetchers have different strengths and

weaknesses, in this chapter, we propose a prefetch mechanism that attempts to combine the strengths of each. More specifically, we propose the Array Tracking Prefetcher (ATP) [1] which tracks array-based indirect memory accesses such as $A[B[i]]$, $A[B[C[i]]]$, $A[B[i][j]]$, and $A[func(B[i])]$ where $func()$ comprises some arithmetic and binary operations, and combinations of these array-based indirect memory access types. Rather than using software to insert prefetching instructions, ATP uses special metadata instructions to pass data structure traversal information to the hardware component. These metadata instructions execute outside of the loop, so they do not result in significant instruction overhead. ATPs hardware component uses the data structure traversal information to configure itself, based on the behavior of the software which can significantly reduce the training time. Furthermore, passing this traversal information, ATPs hardware component does not need to detect the indirect memory access behavior itself, which simplifies the complexity of the hardware while still enabling it to prefetch a wide variety of indirect memory accesses.

Across a set of memory-bound benchmarks, for a single-core architecture, ATP achieved an average speedup of 2.17X (with a maximum of 5.57X) over the no prefetching baseline. By comparison, software prefetching (using manually-inserted and hand-tuned prefetching instruction) had an average speedup of 1.84X while a state-of-the-art indirect hardware prefetcher (IMP) had an average speedup of 1.32X only. For a 4-core architecture, ATP had an average speedup of 1.85X (up to 5.02X) while software prefetching and IMP had average speedups of 1.60X and 1.25X, respectively.

3.3 Array Tracking Prefetcher (ATP)

ATP is an integrated software/hardware approach to prefetching indirect data access patterns. The remainder of this section describes the software and hardware

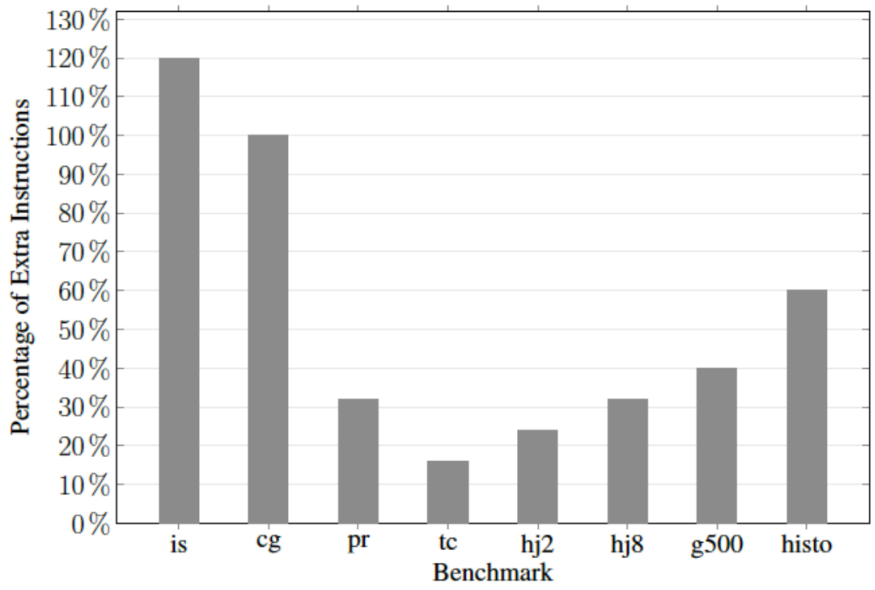


Figure 3.1: Instruction overhead of software prefetching as a percentage of the instructions in the main loop.

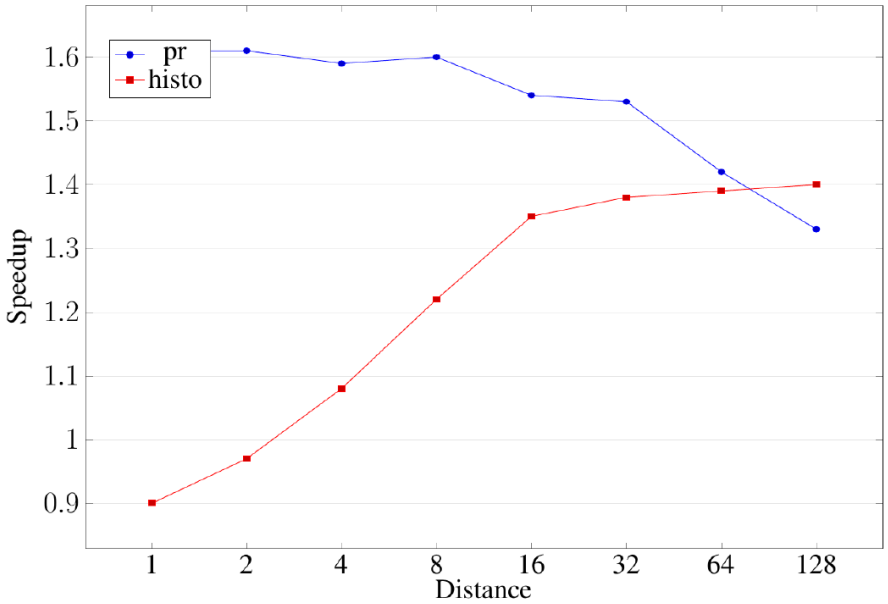
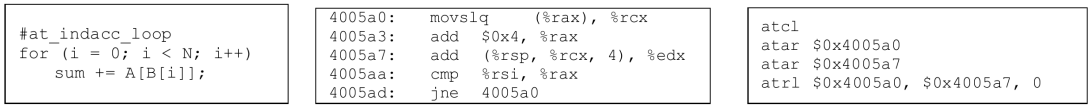


Figure 3.2: Effect of prefetch distance on software prefetching speedup.

components in more depth.



(a) (b) (c)

Figure 3.3: Finding indirect accesses in software. (a) Marking the loop for indirect prefetching. Note that the compiler can automatically perform the marking and generate ATI instructions using a pass similar to approach in [2]. (b) Instructions inside the marked loop. (c) ATI instructions generated for the marked loop (these instructions are placed above the entry point of the loop).

3.3.1 ATP's Software Component

The software component of the ATP extracts information related to indirect memory accesses within a loop and passes this information to the hardware component. The programmer can manually mark this loop as shown in Figure 3.3a or can use the compiler to automatically identify the loop using an approach that is similar to that described in [2]. The software component passes this extracted indirect-access information to the hardware component through special metadata instructions called Array Tracking Instructions (ATIs), as shown in Figure 3.3c.

Array Tracking Instructions (ATIs): Each ATI is a single 6-byte long instruction (two bytes for the opcode, 2-bits to specify the type of the ATI instruction, and the remainder for the operands). When a core detects an ATI instruction, the core removes it from the pipeline and forwards it to the ATP hardware. Because ATI instructions appear only once for each main loop where indirect access traversals occur, the number of executed ATI instructions is insignificant. There are four types of ATIs. $ATCL$ clears all ATP tables. It does not have any operands. $ATAR$ inserts entries into the ATPs Array Table (AT). It has a single operand, a 16-bit offset (from $ATAR$s PC), which is used to compute the PC of the load instruction that accesses the index or target array involved in indirect accesses. $ATRL$ inserts relation information (e.g, between target array A and index array B in an $A[B[i]]$ structure) into the ATPs Indirect Relation Table

Table 3.1: Example sequences of ATI Instructions for different type of indirect array traversals.

Basic (1D index array)	**Pointer (2D index)**	**Multi-level**	**Multi-way (more than one indirect structure)**	**Hashed index**
A[B[i]]	A[B[i][j]]	A[B[C[i]]]	A[B[C[i]]] and A[D[j]]	A[(B[i]-1) & 0xf]
atar pcA atar pcB atrl pcB, pcA, 0	atar pcA atar pcB1 atar pcB2 atrl pcB2, pcA, 0 atrl pcB1, pcB2, 1	atar pcA atar pcB atar pcC atrl pcB,pcA, 0 atrl pcC, pcB, 0	atar pcA atar pcB atar pcC atar pcD atrl pcB, pcA, 0 atrl pcC, pcB, 0 atrl pcD, pcA, 0	atar pcA atar pcB atrl pcB, pcA, 0 atop fsub, 1 atop fand, 0xf

(IRT). It has three operands: the first two are the PCs (offsets) of the load instructions accessing the index and target arrays, respectively. The third operand (1-bit) specifies the type of the relation. 0 is used for $A[B[i]]$ type accesses while 1 is used for $A[B[i][j]]$ type accesses. Finally, $ATOP$ is used for complex data structures where the index to the target array is computed by a function that uses the index array as an argument (i.e., $A[func(B[i])]$). $ATOP$ inserts these operations to ATPs Operation Table (OT). $ATOP$ has two operands. The first operand specifies the operation type (e.g, NOT, ADD, etc.) while the second operand specifies the data for that operation. If there is more than one operation, ATP uses multiple $ATOP$ instructions. The first $ATOP$ instruction always follows an $ATRL$ instruction. The first $ATOP$ instructions first operand is implicitly specified as the index array value (e.g., $B[i]$). The second $ATOP$ uses the result of the first $ATOP$ as its first operand. Table 3.1 shows example sequences of ATI Instructions for different types of indirect array traversals targeted in this work. It is important to note that Access Tracker Unit (ATU) can simultaneously keep track of multiple indirect array access structures with various levels of complexity, which is a significant improvement compared to the state-of-the-art indirect hardware prefetcher, IMP.

Generating ATIs: Generating ATIs consists of two stages. First, the software component generates a dependency graph of the instructions inside a marked

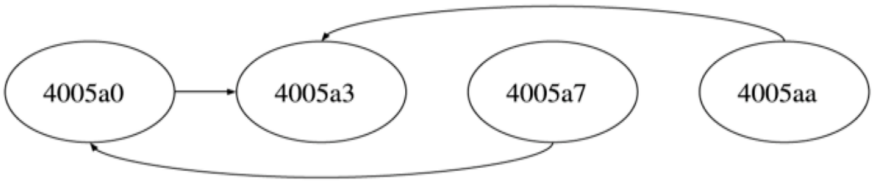


Figure 3.4: Dependency graph generated from the Code Snippet in Figure 3.3b

loop. Figure 3.4 depicts the dependency graph for the code loop shown in Figure 3.3b. Second, the software component generates ATIs based on the type of node or its connections. More specifically, for nodes that correspond to a load instruction, the software component generates an $ATAR$ instruction. The software component generates $ATRL$ instructions (e.g., when one load instruction is used to calculate the address of another load instruction) and ATOP instructions (if the index for the outer array requires computation) based on the connections of the nodes. The software component places these instructions above the entry point of the loop. The software component also places $ATCL$ instructions before the beginning of the loop in order to clear the ATP tables before the loop begins to execute.

3.3.2 ATP Hardware Mechanism

Overview: The hardware component of the ATP uses the information provided by ATIs to initialize the ATP tables. During the execution of the loop, the ATP calculates the size of the array type (i.e., the prefetching stride) and the base address of the required arrays, e.g., the address of $A[0]$ for a $A[B[i]]$. After calculating the strides and base addresses, the ATP starts generating prefetch addresses for forthcoming indirect memory addresses based on the calculated base address and stride. The ATP also includes a mechanism to dynamically change the prefetch distance in order to adapt to specific run-time behavior to achieve better timeliness and performance.

An overview of the ATP hardware mechanism is shown in Figure 3.5. It

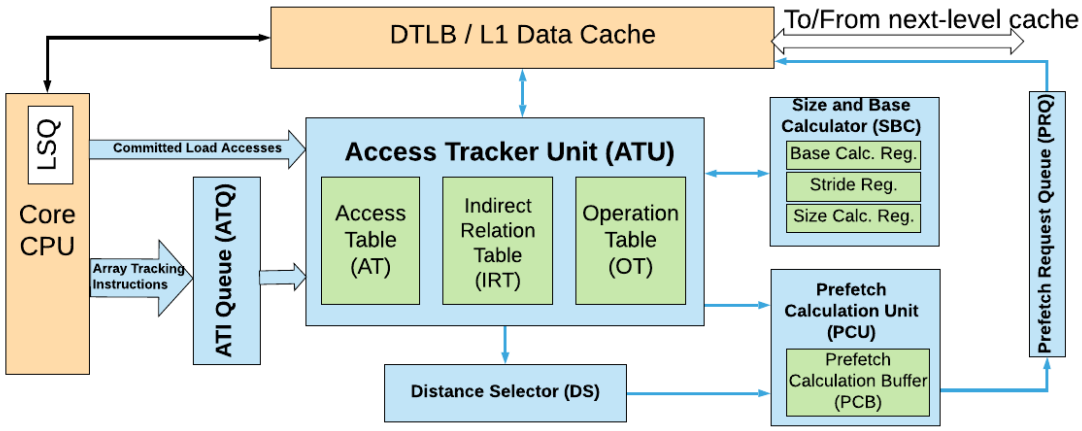


Figure 3.5: An overview of ATP

consists of an ATI queue (ATQ), an Access Tracker Unit (ATU), a Size and Base Calculator (SBC), a Prefetch Calculation Unit (PCU), and a Distance Selector (DS). After an ATI instruction has been identified in the processor pipeline, it is forwarded to the ATQ, which is a FIFO queue with head and tail pointers. A sub-opcode field identifies individual ATI instructions. The ATQ is simply the interface between the processor pipeline and the ATP. ATPs prefetch generation consists of four stages: A) First, ATIs in the ATQ are processed and used to initialize the ATU tables. B) Once all ATIs have been processed, SBC starts calculating the sizes and base addresses for trigger and target arrays, which is needed for prefetch calculation. C) Then, whenever the ATU observes a demand access to a trigger array, it notifies the PCU to begin the prefetch calculation process and issue prefetches. D) Finally, the prefetch distance is dynamically adjusted based on the feedback from the DS, which finds the best performing prefetch distance using a simple mechanism. Next, we explain the operation of ATP in these four stages.

Processing of ATI instructions and ATP Initialization

Each valid ATI in the ATQ is processed in-order from ATQs head to tail. ATIs are used to initialize/program the ATU tables. The ATU consists of three important tables, the Array Table (AT), the Indirect Relation Table (IRT) and the Operation Table (OT). An $ATCL$ instruction resets all the ATU tables, namely valid bits are set to zero in the AT, IRT, and OT. We explain how ATI instructions initialize or program ATU tables using the example in Figure 3.6, which shows the final status of the AT, IRT and OT after they are initialized for $A[(B[i]\&0x7F)*14]$ structure. The indirect access structure in this example generates two $ATAR$ instructions, one for array A and one for array B. The $ATAR$ instructions update the AT. Each $ATAR$ instruction reserves the next available entry in the AT. It specifies a load PC that is involved in reading an array element (and involved in indirect access). The fields in the AT is shown in Figure 3.6. A detailed description of each field in the ATU tables is given in Table 3.2. Initially, trigger-bit and depth field of the AT is 1. Trigger type, indirect map, and root fields are all initially 0s.

For the example in Figure 3.6, following the two $ATAR$ instructions is an $ATRL$ instruction specifying the relation between arrays A and B. Each of the PCs specified by $ATRL$ has already been placed in the AT due to prior $ATAR$ instructions. When an $ATRL$ instruction is executed, it allocates an entry in the IRT, locates the index arrays PC (B) in the AT and updates the indirect map field of the AT entry with the index of the IRT entry it allocated. Then, it locates the target array PC (A) in the AT and saves its index in the destination field in the IRT entry. Indirect map field of the AT is a bitmap (each bit refers to an index of IRT entry) specifying if an IRT entry in relation to the array in the current AT entry exists. 00 means no relation exists and thus array in that AT entry is not used as an index for any target array. In Figure 3.6, ATs entry 1 for array B has

a non-zero indirect map, 10, suggesting the 0th entry in the IRT table provides in its destination field a pointer to the target array (in the AT) for which array B is used as an index for. **If all indirect maps are 0, ATP acts as a stream prefetcher.** This will happen when no $ATRL$ instruction is observed for $ATAR$ instructions.

Level (or depth) of an indirect access depends on the number of indirect accesses made in a chain starting with the access to the index array. AT has a field, depth, monitoring this level. $ATRL$ updates the depth field of the base array (B) by checking if it is less than the depth of the target array, A. If so, it sets the depth of array B to be one more than its target array (in this example, 2). The index array has the highest depth and a target array which is not an index to another target array has the lowest depth, 1. Depth is used for prefetch address calculation as described in Section 3.3.2.

Finally, $ATRL$ is followed by $ATOP$ instructions since the base array is not directly used as an index for the target array. The first $ATOP$ is an AND with data $0x7f$ and the second $ATOP$ is a MUL with 14 as its data. $ATOP$ instructions can only follow an $ATRL$ or another $ATOP$ instruction. $ATOP$ sets to 1 the op bit of the IRT entry it corresponds to denoting that base array must undergo an operation before used as an index for target array. op_idx field specifies the index of the OT that corresponds to the operation specified by $ATOP$. If $ATOP$ is followed by another $ATOP$, the next bit field of the last $ATOP$ is set to 1. Once the last $ATOP$ in the ATQ have been processed, ATU has completed initialization and ATP is ready to move to the size and base calculation stage.

Size and Base Address Calculation

Before prefetching can start for indirect accesses, the stride of the trigger arrays and, the element sizes and base addresses of the target arrays must be

Array Table (AT)												
valid	load_pc	base_bit	base_addr	trigger_bit	trigger_type	size_bit	size	indirect_map	depth	root_bit	root_addr	root_size
1	PC_A	1	Base A	0		1	8	00	1	0		
1	PC_B	0		1	1	1	4	10	2	0		
0												
0												

Indirect Relation Table (IRT)				
valid	destination	type	op_bit	op_idx
1	0	REG	1	0
0				

Operation Table (OT)				
valid	op	data	next_bit	next_idx
1	AND	0X7f	1	1
1	MUL	14	0	

ATI Instructions	
atar	PC_A
atar	PC_B
atrl	PC_A, PC_B, 0
atop	fAND, 0x7f
atop	fMUL, 0xe

Figure 3.6: The final state of the AT, IRT and OT when they are initialized for a $A[(B[i]\&0x7f) * 14]$ structure. A is an array of 8-byte doubles and B is an array of 4-byte integers.

Table 3.2: Detailed Explanation of the fields in the AT, IRT and OT.

Field Name	**Detailed Explanation**
AT/IRT/OT.valid	Specifies if the entry is valid or not. Initial value is zero.
AT.pc	The PC of the load instruction that accesses an array element involved in indirect access. This could either be index or target array access PCs.
AT.base_bit	Set when (and if) base address calculation has been completed.
AT.base_addr	Start (base) address of an indirectly accessed array. For example, for $A[B[i]]$ structure, base address is needed for array A (address of $A[0]$).
AT.trigger_bit	Set if array in this entry is a trigger array. A trigger array must be an index array where its value is used as index to a target array. For multi-level structures, the trigger array is the highest depth (inner most) array.
AT.trigger_type	Two types of trigger accesses are recognized by ATP: regular (type 0) and pointer (type 1). Regular refers to a direct access as in $A[B[i]]$. Pointer triggers refer to two-dimensional index array structures, such as $A[B[i][j]]$.
AT.size_bit	Set when size calculation has been completed.
AT.size	Size (in bytes) of the array element. This information can be supplied by software. It is, however, easy to compute it at-run-time also.
AT.indirect_map	A bitmap, where each bit refers to a specific IRT entry holding the arrays indirect relation to its target array. If multiple bits are set in the bitmap, this array is used as index for more than one target array.
AT.depth	Level of an indirect access. $A[B[i]]$ is a two-level indirect access (depth of A and B are 1 and 2, respectively), whereas $A[B[C[i]]]$ is a three-level indirect access (depth of A, B, C are 1, 2, and 3, respectively). Depth is used for indirect prefetch calculation.
AT.root_bit	Set when root address has been calculated.
AT.root_addr	Used only for trigger type 1 entries. Updated dynamically during prefetch calculation when necessary.
AT.root_size	Size (in bytes) of the root array element size.
AT.node_bit	Set when (and if) it is an access to the first node of a linked list.
AT.node_offset	Offset value from the current node address to the next nodes pointer.
AT.num_nodes	Number of expected nodes in each linked list.
IRT.destination_idx	The index of the AT entry that holds the target array for this base (index) array. For $A[B[i]]$, destination idx is the index of the AT that holds array A.
IRT.type	Specifies the indirect relation type. Two types are supported: direct and pointer. In direct relation, the value read from the base (index) array is used as an index or used in calculation of the index for the target array. Figure 3.3 demonstrates a direct type. Pointer types are used to connect type 0 trigger entries of base arrays to type 1 trigger entries of destination arrays. In pointer type, value read from the base array is used as the root address of destination access. Root addresses are used to calculate prefetch addresses for incoming dimensions in a two-dimensional array.
IRT.op_bit	Set if operations need to be performed on index array for target array index calculation (See Figure 3.6 as example).
IRT.op_idx	The index of the OT entry that specifies the operation to perform.
OT.op	Operation to perform on index array values.
OT.data	Data needed for operation. The first operand is the index array value if previous entrys next_bit is zero. Otherwise, the first operand is the result of the previous operation.
OT.next_bit	Set if another operation (specified in the next OT entry) needs to be performed after the current operation.

known. ATP employs a single mechanism to compute sizes and base addresses.

Size Calculation: For each committed load instruction, if its PC is found in the AT table, and if the size bit is zero, ATPs Size and Base Calculator (SBC) starts the process for size computation. First, the stride between two accesses of the trigger array is computed. The SBC uses a stride register to keep track of trigger array accesses. A stride register consists of a valid bit, an index to the AT (holding

the trigger array), last address, stride and confidence fields. If the observed stride (the difference between addresses of two consecutive accesses) repeats, confidence counter is incremented. If confidence reaches a certain threshold, the detected stride is saved in the AT entry corresponding to the trigger array and its size bit is set. The stride register is then released to be used by other trigger arrays.

The size calculation for a non-trigger array uses a different method. If the size bit is not set for a non-trigger array in the AT, SBC employs a size calculation register (SCR) for that AT entry, which monitors the values read by index arrays and the resulting addresses of the non-trigger array. SCR consists of a valid bit, two AT index fields, two address fields, two computed index (idx) fields, confidence and size fields. SCR holds the AT index or indexes that hold the index array/s for the target array (the non-trigger array). For each committed load, if the index arrays were accessed, the values read from these arrays are recorded in the computed index field (idx) of the SCR and if the OT has any entry for the relation related to that index array, the operation/s are performed on these values to compute the final index (idx1) for the target array and computed index field of SCR is updated. When an access to the non-trigger array is observed (after access to its index array), its address (addr1) is recorded in the SCR. Once SCR observes two addresses (addr1 and addr2) and two indexes (idx1 and idx2), the size of the non-trigger array is computed using Equation (3.1). The size computation is repeated multiple times until a certain confidence threshold is reached. After size is computed, it's recorded in the AT entry corresponding to the non-trigger array, and SCR is released.

$$Size(A) = \frac{Addr(A[B[i+1]]) - Addr(A[B[i]])}{B[i+1] - B[i]} \tag{3.1}$$

Base Address Calculation: Finally, before indirect prefetching can be per-

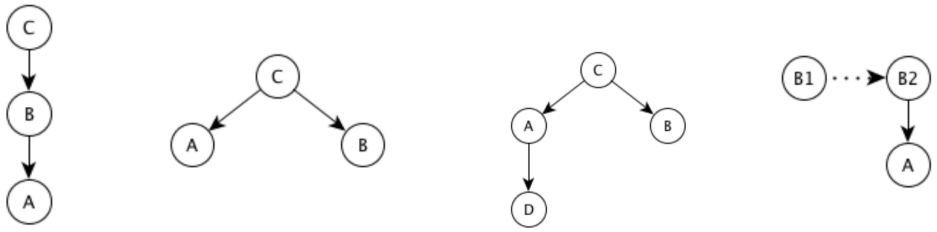
formed for target arrays, their base addresses must be computed. SBC assigns a base address calculation register (BACR) to a non-trigger type entry of a destination array if the base address of it is not yet calculated but the size of it is already known. The value of the idx field in the BACR is set to the index of the AT entry which it is assigned for. Like the SCR, BACR also has a field for the indirect map which shows the entries of source (index) arrays in the AT.

After BACR is assigned to an entry in the AT, SBC monitors the accesses requested for any of the source arrays by checking its indirect map field. When SBC is notified by an access to a source array (e.g., $B[i]$ in Equation (3.2)), it stores the value of the accessed data in its BACR after performing the required operations in the OT if any operation exists between the source array and the destination array (value). Once an access to the destination array (addr) is seen by SBC, it calculates the base address based on Equation (3.2) (here $B[i]$ is the value and $Addr(A[B[i]])$ is the addr). If the same base address is computed multiple times to satisfy a confidence threshold, SBC sets the base address field of the corresponding entry in the AT and releases the BACR.

$$Addr(A[B[i]]) = BaseAddr(A) + (B[i]xSize(A)) \tag{3.2}$$

Prefetch Triggering Calculation

As shown in Figure 3.5, ATP employs a Prefetch Calculation Unit (PCU) to calculate prefetch addresses when triggered. Prefetch calculation in PCU is performed in two steps: 1) prefetch initialization and 2) prefetch address calculation and issuing of prefetches. Upon an access to a trigger array the ATU signals the PCU to start a prefetching operation. The PCU begins the initialization phase by first allocating an entry for the trigger array in the Prefetch Calculation Buffer (PCB). The PCB is a temporary buffer that keeps detailed information about the



A[B[C[i]]] A[C[i]] and B[C[i]] D[A[C[i]]] and B[C[i]] A[B[i][j]]

(a) (b) (c) (d)

Figure 3.7: Graph representation of different indirect access behaviors.

entries of potential prefetches; it is cleared after all computed prefetches are issued. By referring to the ATU tables, the target array/s characteristics are also inserted into the PCB and they are linked to their sources in the PCB entries. After the initialization is done, the PCU starts calculating the prefetch addresses for each entry.

ATP can efficiently calculate prefetch addresses for many different indirect access structures and their combinations. Figure 3.7 represents each indirect access behavior using a graph model where each entry in the AT is a node and each entry in the IRT is an edge. Since this graph model is in the form of a tree structure, the root of the tree represents the index (trigger) array. A prefetch calculation is performed for each node of this graph. Here, we explain how prefetching is performed for each indirect structure that is shown in Figure 3.7.

In a multi-level $A[B[C[i]]]$ structure, the index array, C, represents the root node of the graph as shown in Figure 3.7a. As ATU signals PCU for prefetching operation on an access to the trigger array C, an entry for C is allocated in the PCB. Since C is the trigger array, by following the indirect map fields in the AT, its target array B and then A are also inserted into the PCB and they are linked to their sources in the PCB entries. After the initialization is done, the PCU starts calculating the prefetch addresses for each entry. To calculate the prefetch address

for any non-trigger array, the PCU needs to read a value from the source array. Assuming that we want to calculate a prefetch address for $A[B[C[i+dist]]]$ triggered by an access to $C[i]$, we need to read the value of $C[i+dist]$ and then $B[C[i+dist]]$ to be able to calculate the prefetch address for $A[B[C[i+dist]]]$. Naturally, to be able to find these (source) values in the cache when needed, they need to be prefetched ahead of time. So for an $A[B[C[i]]]$, a depth-3 indirect structure, PCU calculates prefetch addresses and performs prefetching for $C[i+3*dist]$, $B[C[i+2*dist]]$ and $A[B[C[i+1*dist]]]$. In general, an access to a trigger array initiates as many prefetches as there are levels in the indirect structure unless the prefetch address computation fails due to cache misses. Table 3.3 shows the prefetch calculation steps of an $A[B[C[i]]]$ access structure. Events under the same step for different paths may occur in parallel. For instance, in step1: for path C, we can compute prefetch address for $C[i+3*dist]$; for path $C \rightarrow B$, we can read $C[i+2*dist]$; and for path $C \rightarrow B \rightarrow A$, we can read $C[i+1*dist]$.

As Table 3.3 shows, PCU takes the following actions (triggered by an access to $C[i]$) to perform prefetching for an $A[B[C[i]]]$ structure.

For node C:

Step 1. Compute the prefetch address for $C[i+3*distance]$ using Equation 3.3 and issue the prefetch.

For node B:

Step 1. Compute the address for $C[i+2*distance]$ using Equation 3.3 and read its value from the L1 cache.

Step 2. Compute the prefetch address for $B[C[i+2*distance]]$ using Equation 3.2 and issue the prefetch.

Table 3.3: Prefetch calculation steps of $A[B[C[i]]]$ structure.

Path	Step 1	Step 2	Step 3
C	**Prefetch** $C[i+3*dist]$		
$C \rightarrow B$	Read $C[i+2*dist]$	**Prefetch** $B[C[i+2*dist]$	
$C \rightarrow B \rightarrow$ A	Read $C[i+1*dist]$	Read $B[C[i+1*dist]$	**Prefetch** $A[B[C[i+1*dist]]]$

Table 3.4: Prefetch calculation steps of $A[C[i]]$ and $B[C[i]]$ structure.

Path	Step 1	Step 2	Step 3
C	**Prefetch** $C[i+3*dist]$		
$C \rightarrow A$	Read $C[i+2*dist]$	**Prefetch** $B[C[i+2*dist]$	
$C \rightarrow B$	Read $C[i+1*dist]$	Read $B[C[i+1*dist]$	**Prefetch** $A[B[C[i+1*dist]]]$

For node C:

Step 1. Compute the address for $C[i+1*distance]$ using Equation 3.3 and read its value from the L1 cache.

Step 2. Compute the address for $B[C[i+1*distance]]$ using Equation 3.2 and read its value from the L1 cache.

Step 3. Compute the prefetch address for $A[B[C[i+1*distance]]]$ using Equation 3.2 and issue the prefetch.

$$PfAddr = CurrAddr + (SizexDepthxDistance) \tag{3.3}$$

Figure 3.7b shows a graph generated from the multi-way indirect access structure of A[C[i]] and B[C[i]]. Since the depth of the tree is 2 in this example, prefetch calculation can be done in two steps as shown in Table 3.4. The indirect access structure of D[A[C[i]]] and B[C[i]] (both multi-level and multi-way) generates a graph with a depth of 3 as shown in Figure 3.7c. The steps of prefetch calculation for this behavior is shown in Table 3.5. The PCU operation (both initialization and prefetch calculation) for the A[func(B[i])] structure is similar; thus, it is not presented.

Table 3.5: Prefetch calculation steps of $D[A[C[i]]]$ and $B[C[i]]$ structure.

Path	Step 1	Step 2	Step 3
C	**Prefetch** $C[i+3*dist]$		
$C \rightarrow A$	Read $C[i+2*dist]$	**Prefetch** $A[C[i+2*dist]$	
$C \rightarrow B$	Read $C[i+2*dist]$	**Prefetch** $B[C[i+2*dist]$	
$C \rightarrow A \rightarrow D$	Read $C[i+1*dist]$	Read $A[C[i+1*dist]$	**Prefetch** $D[A[C[i+1*dist]]]$

Finally, Figure 3.7d shows the graph representation of the indirect access structure of $A[B[i][j]]$, which we classify as having a pointer-type relation. The index array in this structure is a two-dimensional array and it is represented by two separate nodes (i.e., separate entries in the AT). Node $B1$ represents the first dimension of array B (i.e., the load instruction that reads $B[i]$) and node $B2$ represents the second dimension (i.e., the load instruction that reads $B[i][j]$). Also, the edge from $B1$ to $B2$ is represented as a pointer type relation instead of a regular type as for the structures discussed above (for Figure 3.7a-3.7c).

In two dimensional arrays, prefetching can be triggered by two different load accesses ($B1$ and $B2$). Table 3.6 shows the steps of prefetch calculation in an indirect access structure of $A[B[i][j]]$. Prefetch address calculation is somewhat more complicated for this structure. When ATU observes an access to $B1$ entry, the prefetch address for $B1$ is calculated as usual. However, prefetch addresses for $B2$ and A are not calculated at this moment as the relation type between $B1$ and $B2$ is a pointer type. Instead, the root address field of $B1$ entry in the AT table is updated with the value read by the current access (the value of $B[i]$, which is the address of $B[i][0]$). Prefetching for entries $B2$ and A are triggered by the access to the $B2$ entry. As it is shown in Table 3.6 when prefetching is triggered by an access to $B[i][j]$, ATP calculates the addresses of $B[i+2*dist][j]$ and $A[B[i+1*dist][j]]$. To be able to calculate the prefetch address for $B[i+2*dist][j]$, we need to read the address of $B[i+2*dist][0]$ (i.e., the value of $B[i+2*dist]$) which we call the prefetch root address. Using the root address (address of $B[i][0]$) which was

Table 3.6: Prefetch calculation steps of $A[B[i][j]]$ structure.

Path (1st dimension)	Step 1	Step 2	Step 3
$B1$	**Prefetch** $B[i+3*dist]$ Set root addr in the AT to $B[i]$		
Path (2nd dimension)	Step 1	Step 2	Step 3
$B2$	Read $B[i+2*dist]$	**Prefetch** $B[i+2*dist][j]$	
$B2 \rightarrow A$	Read $B[i+1*dist]$	Read $B[i+1*dist][j]$	**Prefetch** $A[B[i+1*dist][j]]$

saved previously by trigger $B1$, we calculate the address of $B[i+2*dist]$, in which the prefetch root address is stored, by using the first part of Equation 3.4. Then, we can calculate the prefetch address (address of $B[i+2*dist][j]$) using the second part of Equation 3.4. This calculation is based on the fact that the distance between $B[i][0]$ and $B[i][j]$ should be equal to the distance between $B[i+2*i][0]$ and $B[i+2*i][j]$. Finally, prefetch address calculation for $A[B[i+1*dist][j]]$ is done in a similar way: it calculates the address of $B[i+1*dist][j]$; read its value and uses it to calculate the prefetch address for $A[B[i+1*dist][j]]$ as shown in Table 3.6.

$$\begin{aligned} PfRootAddr &= RootAddr + (SizexDepthxDistance) \\ PfAddr &= Mem[PfRootAddr] + (AccessedAddress - Mem[RootAddr]) \end{aligned} \tag{3.4}$$

Adaptive Prefetching Distance Selection

Distance Selector (DS) enables ATP to adjust the prefetch distance (in terms of how many array elements ahead) dynamically to be timely accurate on different applications and configurations.

Each power of 2 prefetch distance from 2 to 16 competes during a test period and at the end of this period, the distance that takes the smallest number of cycles to complete the same number of loop iterations is picked as the distance for the acting period that comes after the testing period. The acting period is a fixed 50 times larger than the testing period. After acting period another testing period follows.

In testing period, each prefetch distance is run for a fixed number of loop iterations (64 in our experiments of which the first 32 are used for warm-up and next 32 are used for performance measurement) in a round-robin fashion and the number of cycles is counted. DSU employs two 32-bit cycle counters. *min_count* holds the smallest cycle count and *run_count* holds the cycle count for the currently tested distance. After each distance has completed its test, if $run\_count < min\_count$, *min_count* is set to *run_count* and a 3-bit *best_dist* register is updated. After all of the distances are tested, *best_dist* indexes a table of 2-bit confidence counters and increments the count for that distance. DSU repeats this process until any of the distances confidence reaches to a threshold which is 2 in our implementation. Using a threshold less then 2 decreases the performance of some applications due to aggressive decisions. Using a threshold above 2 proved useless and increases the duration of the testing phase which also decreases the performance. Once the decision is made, the chosen distance is set to be used in the acting period and the testing period cycle counters are set to 0.

In multi-core architectures, distance selection is performed separately on each core. We observe that best distances vary for each core running a multi-threaded application due to the sharing of last-level cache and memory bandwidth.

3.3.3 Extending ATP for Prefetching Linked Data Structures

ATP is very successful in prefetching for indirect access structures as we show in the results section. However, for one of our workloads, HJ8, a significant performance opportunity was lost because indirect accesses were followed by linked list traversals and ATP was not able to capture this behavior. In HJ8, each element of the destination array is a linked list data structure. We extended the ATP to support linked lists. We add three additional fields in the AT: a node bit, node offset and number of nodes. An additional instruction called *atnod* is also added

to inform the hardware of this behavior and set the fields in the AT. This instruction always follows an ATAR instruction and have two operands: an offset that represents the next field in a linked-node and an immediate value that represents the number of nodes. For example, for a $A[B[i]]{->}\,next{->}\,next$ structure, where each element of array A is a head node of a linked list where each list has 3 nodes (a head node and two additional nodes), generated ATI sequence is as follows:

```
atar PC_B
atar PC_A
atnod 0x18, 2
atrl PC_B, PC_A, 0
```

The *atnod* instruction will set the node bit, node offset and the number of nodes fields of the AT table. In this example, *atnod* suggests that each element of the array A points to the head node of a linked list. *atnod*s first operand (offset) shows that $A[B[i]] + 0x18$ points to the next node of the linked list ($A[B[i]]{->}\,next$). The second operand (which is 2 in this example) shows the number of nodes except for the head node in the linked list. Although ATP can stop prefetching nodes if it reaches to the end of the list (where the next node pointer is set to $NULL$), it requires this information to calculate the depth of prefetches in the chain. When the prefetch is triggered by an access to the index array, $B[i]$, ATP uses the same approach that it does for indirect memory accesses to prefetch future linked node accesses. The steps of prefetch address calculation, in this case, are shown in Table 3.7.

We observe significant performance improvement in HJ8 with an ATP that supports linked list traversals. Without this support, ATP achieves a speedup of 1.44 for this benchmark by successfully prefetching the indirect accesses. With the added linked list support, ATP boosts the speedup to 3.32. By contrast, software

Table 3.7: Prefetch calculation steps of $A[B[i]]$ structure where each element of array A is a linked list.

Path	Step 1	Step 2	Step 3	Step4
B	**Prefetch** $B[i+4*dist]$			
$B \rightarrow A$	Read $B[i+3*dist]$	**Prefetch** $A[B[i+3*dist]$		
$B \rightarrow A$	Read $B[i+2*dist]$	Read $A[B[i+2*dist]$	**Prefetch** $A[B[i+2*dist]{->}next$	
$B \rightarrow A$	Read $B[i+1*dist]$	Read $A[B[i+1*dist]$	Read $A[B[i+1*dist]{->}next$	**Prefetch** $A[B[i+1*dist]{->}next{->}next$

prefetching, SWPF, that prefetch for both indirect accesses and the linked list in HJ8 could only achieve a speedup of 1.65, which is much lower than ATPs performance. It is important to note, however, that our implementation is limited to cases where the number of nodes is known. HJ8 has 3 nodes for all the linked lists so prefetch depth is constant (e.g., depth is 4 in the example in Table 3.7) and ATP does not need to predict depth (distance, however, is dynamically evaluated as before). When the number of nodes is not known, ATP must make predictions for the depth of the structure, which complicates the hardware mechanism. This scenario is not evaluated and left as future work since our focus in this chapter is indirect access structures.

3.4 Experimental Setup

We now discuss details of the simulation infrastructure, the workloads and the configurations that we used for our evaluation.

3.4.1 Simulation Environment

We implemented the ATP on the gem5 simulator [5] using System Emulation mode and generated the results using the x86 out-of-order CPU model. Table 3.8 shows the configuration of each core while Table 3.9 shows the ATP configuration. We inserted ATI instructions at the beginning of the loop and implemented ATP to prefetch for the L1 cache in order to provide for a direct comparison with IMP [4], which prefetches for the L1 cache. Each L1 is equipped with an 8-entry prefetch request queue (PRQ) in our evaluation. In all methods tested, computed prefetch

Table 3.8: Simulator Configurations

ISA	64-bit x86
Number of Cores	1-8
Architecture	4-Issue, Out-Order, 2GHz
LQ/SQ Entries	64/36
ROB Entries	168
Branch Pred.	Tournament BP
L1 Cache	Private, 8-way 32KB, mshrs: 8, latency: 4 cycles
L2 Cache	Private, 8-way 256KB, mshrs: 16, latency: 12 cycles
L3 Cache	Shared, 16-way, 1MB per core, mshrs: 16, latency: 32 cycles
Memory	DDR3-1600, 800MHz, 1 channel, 2 ranks, 8 banks/rank, 512K rows/bank, 1kB row, baseline $t_{RCD}/t_{RAS}/t_{WR}$: 13.75/35/15 ns
Memory Controller	64-entry RD/WR request queue, FR-FCFS scheduling, closed-row policy

addresses are placed into the PRQ before they are issued to the L1 cache.

We faithfully implemented IMP (attached to each L1 cache) on our baseline architecture. Similar as in [4], our IMP implementation used a 16-entry Prefetch Table and a 4-entry Indirect Pattern Detector with 4-base address length and 4 shift values. Total hardware budget for our IMP implementation was 8032 bits (1004 bytes) per core, almost four times the size of the ATP (see Table 3.3). To evaluate the performance of software prefetching, we inserted software prefetching instructions inside the loops containing the indirect memory accesses. For both IMP and software prefetching, we measured the speedup for various prefetch distances but only report the results for the best performing distance.

For each benchmark, we fast-forward to the beginning of the loop containing indirect memory accesses and the simulate 100M instructions; for multi-core simulations, each core simulates at least 100M instructions.

We use the number of cycles per loop-iteration as the performance metric as it eliminates additional overhead due to software prefetching. As such, it provides an apples-to-apples comparison between hardware and software prefetching.

Table 3.9: Hardware budget of ATP on single-core architecture.

	ATI Queue	Array Table	Relation Table	Operation Table	Prefetch Table	Stream Register	Size Calc. Reg	Base Calc. Reg.	Distance Selector
# of Entries	4	4	4	2	4	1	1	1	-
Entry size (bits)	80	268	6	39	69	101	172	105	118
Total Size: 2266 bits (∼284 bytes)									

3.4.2 Benchmarks

We used seven benchmarks to evaluate the performance of ATP. Each benchmark contains indirect memory accesses inside their performance-critical loop.

Integer Sort (IS) and *Conjugate Gradient (CG)* are from the NAS Parallel Benchmarks suite [6]. *IS* represents computational fluid dynamics programs and uses a bucket sort algorithm to sort integer values while *CG* represents unstructured grid computations and use eigenvalue estimation on sparse matrices. *IS* and *CG* have simple $A[B[i]]$ access behavior.

Both Pagerank (PR) and *Triangle Counting (TC)* are from the CRONOSuite benchmark suite [7]. PR is a graph algorithm that ranks a website based on the rank of the websites that link to it [8] while TC counts the number of triangles in a graph and is used by graph algorithms such as clustering coefficients [9]. *PR* and *TC* have simple $A[B[i][j]]$ access behavior.

Hash Join [10] hashes the keys stored in an array and uses the hashed values to access another array. Each bucket in the hash table consists of a linked list. We used two different variations of this benchmark: (1) *Hash Join 2EPB (HJ2)* has only one node per bucket and (2). *Hash Join 8EPB (HJ8)* has three nodes per bucket; as such it performs memory accesses for the additional nodes. Hash Join is a kernel representative of database applications.

Graph500 [11] (g500) runs a breadth first search (BFS) algorithm over a graph data structure. It performs indirect memory accesses while accessing neighbor vertices.

Histogram (Histo) calculates the distribution of numerical data and is from the Parboil benchmark suite [12].

3.5 Results

This section presents the performance of ATP, software prefetching (SWPF), and IMP, which is a pure hardware prefetching mechanism. We measure the performance of ATP, SWPF, and IMP for single and multi-core architectures. **It is important to note that the results are biased in favor of SWPF because** we presented the best speedup achieved by SWPF after carefully inserting prefetches and many profiling runs to obtain the best performing prefetch distances.

3.5.1 Single-Core Performance of ATP

Figure 3.8 shows the speedup of ATP, SWPF, and IMP over the no-prefetching baseline architecture. The average (geometric mean) speedup of ATP is 2.17 which outperforms both SWPF (1.84) and IMP (1.32). For IS, SWPF and ATP both outperform IMP, while ATP and IMP outperform SPWF for *CG*. As described in Section 3.2, the overhead due to SWPF was extremely high for both *IS* and *CG*. This overhead has little effect in *IS* because SWPF can hide this overhead by virtue for significantly reducing the latencies of the indirect memory accesses.

CG is one of the most sensitive benchmarks to the instruction overhead of software prefetching. Using software prefetching for *CG* decreases its performance by 33% while hardware prefetching mechanisms can achieve better performance (1.40 and 1.60 for ATP and IMP respectively). For *CG*, ATP has a lower speedup compared to IMP. Our evaluations show that for the same fixed distance value, ATP and IMP have a similar result for *CG*. However, ATPs distance selector does not always use the best performing distance for *CG*, which in turn impacts the

overall performance potential negatively.

For *PR* and *TC*, SWPF and ATP outperform IMP. More specifically, for *PR*, SWPF and ATP have speedups of 1.53 and 1.20, respectively, while IMP has a 1.07 speedup. For *TC*, ATP and SWPF have a speedup of 1.76 and 1.65, respectively, while IMP does not have any speedup. *PR* and *TC*'s $A[B[i][j]]$ type of indirect memory accesses which is challenging for IMP as prefetching for the outer loop (i.e., a prefetch distance of $i + distance$) yields more speedup than prefetching for the inner loop (i.e., $j + distance$). By contrast, both SWPF and ATP are able to adjust the prefetch distance to maximize speedup although the overhead associated with SWPF degrades its achievable performance. However, SWPF still outperforms ATP for the PR benchmark.

For *HJ2* and *HJ8*, ATP and SWPF again outperform IMP. More specifically, for HJ2, ATP and SWPF have speedups of 5.57 and 5.40, respectively, while IMP provides effectively no speedup. For *HJ8*, ATP and SWPF have speedups of 3.32 and 1.67, respectively, while IMP again provides effectively no speedup. IMP provides effectively no speedup over the no-prefetching baseline because IMP has great difficulty detecting the $A[func(B[i])]$ type of indirect memory accesses present in *HJ2* and *HJ8*. The speedups of ATP and SWPF in *HJ2* are higher than those in *HJ8* because the latter contains linked list accesses which increases the depth of the access structure. ATP has a significant advantage over SWPF here since SWPF adds more instruction overhead while ATP can prefetch the linked list nodes without any extra instruction penalty.

Finally, as shown in Figure 83.8, all methods perform similarly for *g500*. ATP and SWPF have a slightly better speedup (1.22 and 1.23 respectively), while IMP has a 1.18 speedup for this benchmark.

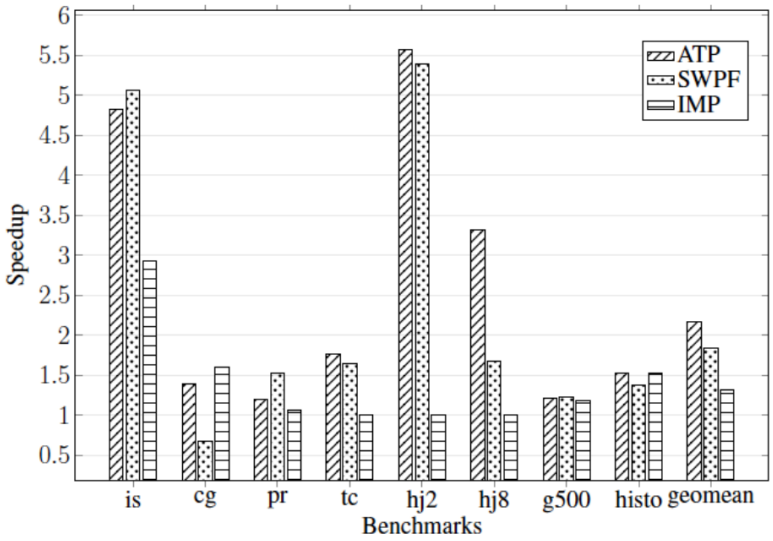


Figure 3.8: Performance comparison of SWPF, IMP and ATP on single core architecture.

3.5.2 Multi-Core Performance of ATP

Figures 3.9 and 3.10 show speedups due to ATP, SWPF, and IMP on 4-core and 8-core architectures, respectively. Overall, for 4-core architectures, ATP has the highest average speedup (1.85) followed by SWPF (1.60) and IMP (1.25). For 8-cores, speedups for ATP, SWPF, and IMP are 1.41, 1.30, and 1.07, respectively.

For *g500*, benefit from prefetching increases on multi-core architectures. SWPF slightly outperforms ATP for this benchmark because ATP loses prefetch opportunities due to dropped prefetches (as we discuss in Section 3.5.6). Generally, the speedups due to prefetching in 4 and 8-core architectures are lower than on a single-core architecture due to increased resource utilization. By way of example, for *IS*, because the main loop is not long enough to hide memory latency, the speedup in *IS* decreases due to an increased number of memory accesses and the concomitant increase in latency for those accesses. Therefore, while resource contention has some effect on the efficacy of each prefetching method, ATP and SWPF still provide significant speedup.

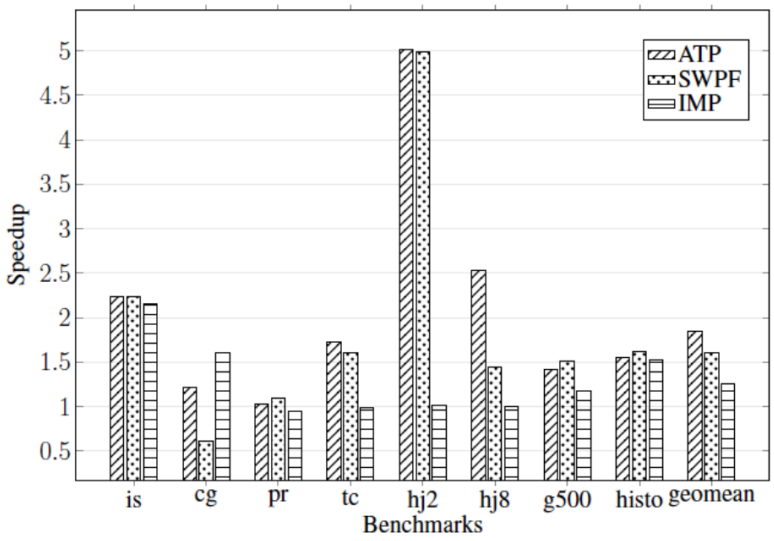


Figure 3.9: Performance comparison of SWPF, IMP and AT on 4-core architecture. Baseline is 4-core architecture with no prefetching.

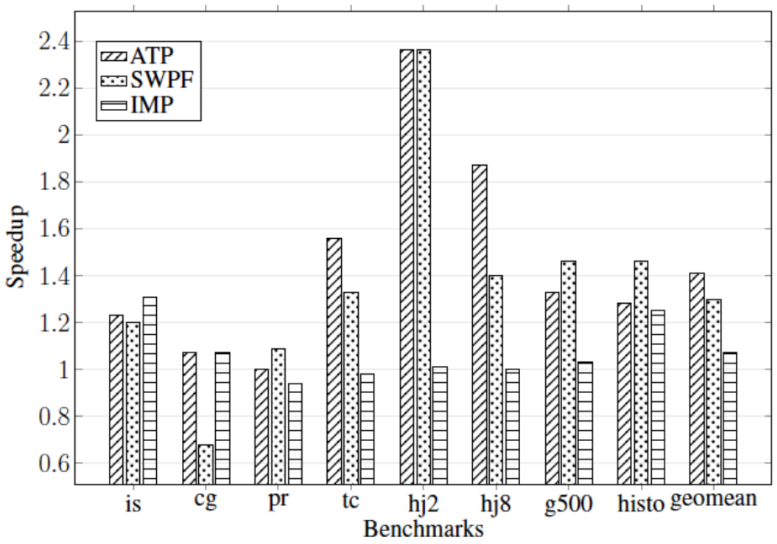


Figure 3.10: Performance comparison of SWPF, IMP and AT on 8-core architecture. Baseline is 8-core architecture with no prefetching.

3.5.3 Efficacy of Adaptive Distance on ATP Speedup

As described above, the Distance Selection Unit allows ATP to dynamically adjust the prefetch distance for different applications and configurations. Figure 3.11 compares the speedup of ATP when using adaptive prefetch distance versus ATP with various fixed distances (2, 4, 8, 16, and 32). The results in Figure 3.11

show that dynamically adjusting the prefetching distance has an average speedup of 2.17 while the highest performing fixed distance ($distance = 8$) yields an average speedup of 1.88. Therefore, even though periodically testing each distance for a certain number of iterations to choose the best distance for the next period may slightly degrade the speedup for some benchmarks, on average, dynamically adjusting the prefetch distance yields a higher average speedup.

IS and *HJ2* benefit from longer prefetch distances due to the small number of instructions in its main loop. As such, in order to be timely, prefetch instructions must be issued further away.

By contrast, *PR* and *TC* have higher performance when using shorter prefetch distances as Figure 3.11 shows. The indirect memory access behavior in these benchmarks is of the form $A[B[i][j]]$. Because the second dimension of the array B, i.e., j, is very short (16 for *PR* and 4 for *TC* for the inputs we used), ATP calculates prefetch addresses based on the first dimension instead. This increases the number of total instruction executed between two consecutive accesses to the first dimension of the array B which favors using shorter distances.

3.5.4 Prefetch Coverage and Accuracy

A prefetcher needs to be accurate or it will prefetch memory blocks that are never used, thus polluting its cache. If a prefetcher is not timely, it will either not fully hide the memory latency of the cache miss or, even worse, the prefetched cache line will be evicted. Table 3.10 shows the accuracy and timeliness for different benchmarks using SWPF, IMP, and ATP. Accuracy is the percentage of prefetched cache lines which are accessed later. Timeliness is the percentage of cache hits to previously prefetched cache lines to the total number of accurately prefetched cache lines.

SWPF has 100% accuracy for all benchmarks because it handles border checks

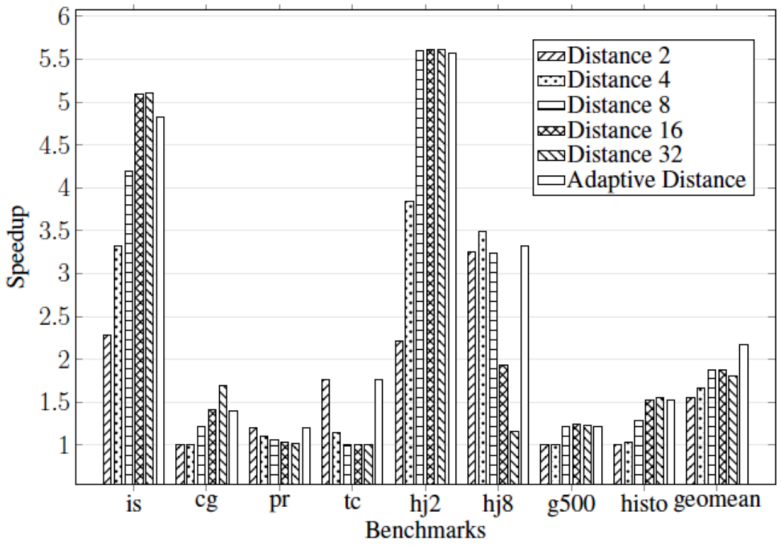


Figure 3.11: Performance comparison of ATP using different fixed distances and adaptive distance adjustment.

Table 3.10: Prefetch accuracy, timeliness, and coverage

Benchmark	ATP			SWPF			IMP		
	Acc.	Tim.	Cov.	Acc.	Tim.	Cov.	Acc.	Tim.	Cov.
is	100%	92%	82%	100%	100%	86%	100%	100%	62%
cg	99%	62%	47%	100%	100%	55%	100%	80%	21%
pr	100%	100%	75%	100%	3%	61%	37%	15%	2%
tc	100%	98%	62%	100%	0%	63%	3%	3%	0%
hj2	100%	100%	99%	100%	100%	100%	100%	100%	4%
hj8	100%	98%	75%	100%	60%	86%	100%	100%	2%
g500	91%	69%	70%	99%	100%	92%	97%	100%	17%
histo	100%	84%	71%	100%	100%	96%	100%	97%	45%
avg	99%	88%	73%	100%	70%	80%	80%	74%	19%

and guarantees the prefetched cache line will be accessed. The prefetch accuracy of ATP and IMP are lower (99% and 80%, respectively). ATP is more accurate than IMP since the software mechanism specifies and limits the prefetches, thus reducing the number of useless prefetches.

We chose the best-fixed distances for SWPF and IMP in our evaluations. The average timeliness for SWPF and IMP is 70% and 74%, respectively. By contrast, ATP dynamically adjusts the distance; the overall timeliness of ATP is

significantly higher (88%). Although SWPF has the best coverage with 80%, ATP also has high coverage with 73% since it calculates the prefetch addresses based on software hints. IMP fails to cover most of the potential prefetches as discussed in Section 3.2 (because it covers a relatively small number of indirect access patterns) and therefore it has much lower coverage (19%).

3.5.5 Effects of Number of MSHRs, L1 Cache Size, and L1 Cache Access Latency

In this section, we evaluated the performance of ATP, SWPF, and IMP as we vary the number of MSHRs, L1 cache size, and L1 cache access latency. Figures 3.12 and 3.13 show the effect of varying MSHRs on the performance of 1-core and 4-core architecture, respectively.

Figure 3.12 shows the overall results using 4, 8, and 16 MSHRs in L1 Cache in the single-core architecture. The increasing number of MSHRs from 4 to 8 and 16 increases the performance of no prefetching baseline by 13% and 16%, respectively (results not shown). Prefetching benefits from the higher number of MSHRs more significantly than the no-prefetching baseline. As shown in Figure 3.12, the number of MSHRs have a big impact on all prefetching methods. The performance of IMP increases by 27% and 40% with 8 and 16 MSHRs, respectively, compared to its performance with 4 MSHRs. SWPF performs 32% better with 8 MSHRs and 57% better with 16 MSHRs. ATP benefits the most from the higher number of MSHRs 42% better with 8 MSHRs and %59 better with 16 MSHRs, compared to 4 MSHRs. For 4-core architecture, as shown in Figure 3.13, the number of MSHRs also has a big impact. Increasing the number of MSHRs from 4 to 8 shows improvements similar to the single-core case. However, 16 MSHRs does not have a significant benefit over 8 MSHRs since prefetching is already limited by higher utilization of shared resources.

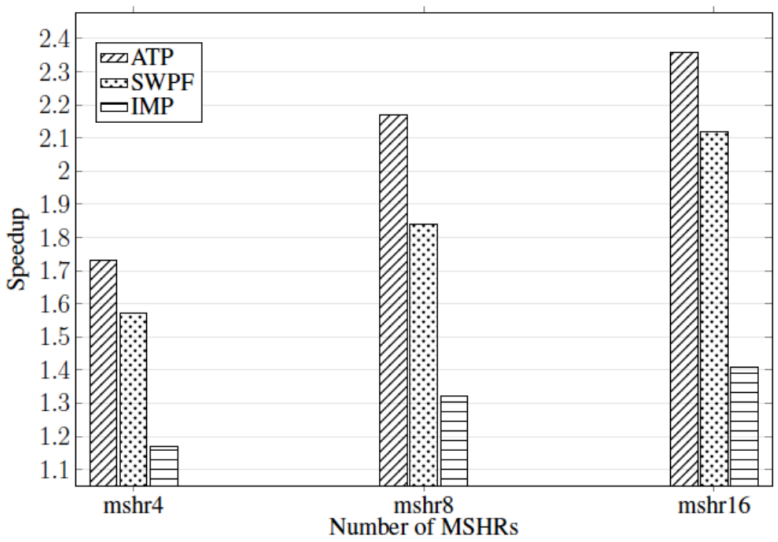


Figure 3.12: Effect of number of MSHRs in single-core architecture.

Figures 3.14 and Figure 3.15 show the overall speedups of all prefetching methods using different L1 cache sizes on single-core and multi-core architectures, respectively. The size of L1 cache has a very limited effect on the speedups either in no prefetching or prefetching methods. Also, both single-core and multi-core results show that using prefetching on a smaller L1 cache size has much better performance compared to using a larger L1 cache size without prefetching. In all cache sizes that we evaluated, ATP outperforms SWPF and IMP.

Figures 3.16 and 3.17 show the effect of L1 data cache access latency on the performance of the prefetching methods that we evaluated. Figure 3.16 shows the effect of three different access latencies on the single-core architecture and Figure 3.17 shows the same results for the multi-core architecture. Smaller cache latencies have a positive impact on all methods as expected. However, the benefit of smaller latencies on prefetching methods reduces on multi-cores due to higher utilization of shared resources.

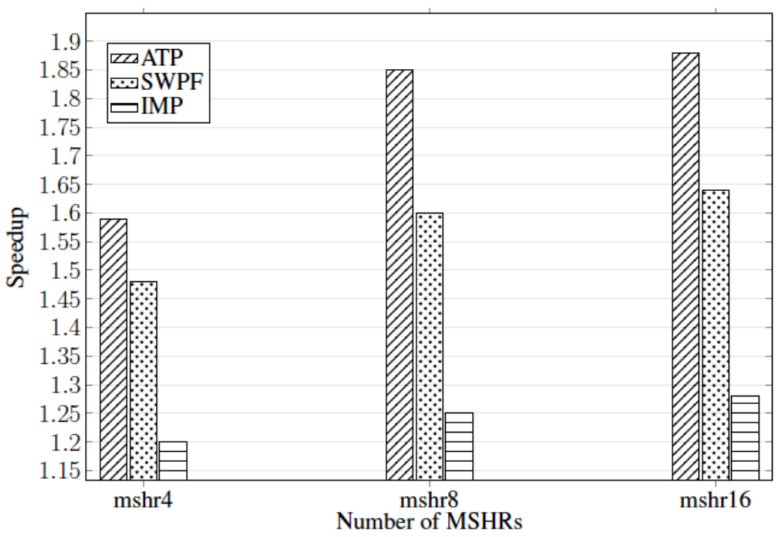


Figure 3.13: Effect of MSHRs in 4-core architecture.

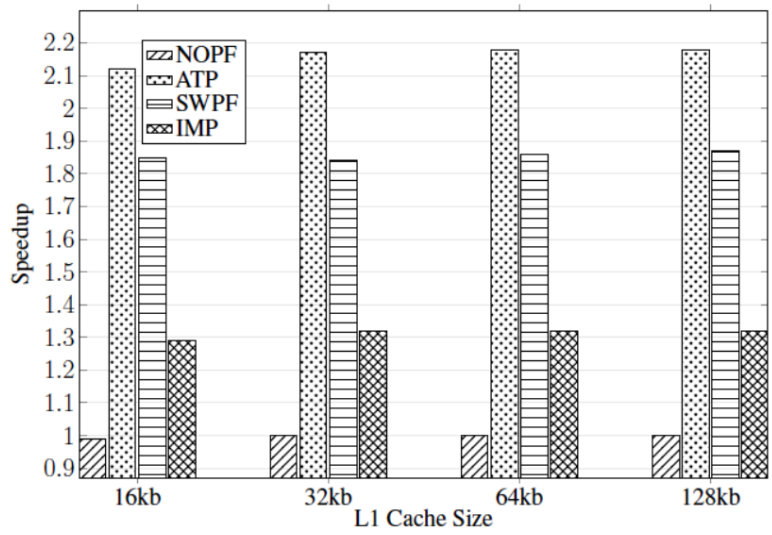


Figure 3.14: Effect of L1 Cache size in single-core architecture. Baseline is no prefetching with 32KB L1 cache.

3.5.6 Prefetch Drops

Prefetch address calculation for indirect memory accesses requires the data of the source arrays to be present in cache at the time of calculation. Failing to read the required data will cause the indirect prefetch address calculation to be dropped. Prefetching the values of index arrays accurately and on-time is critical for indirect prefetching.

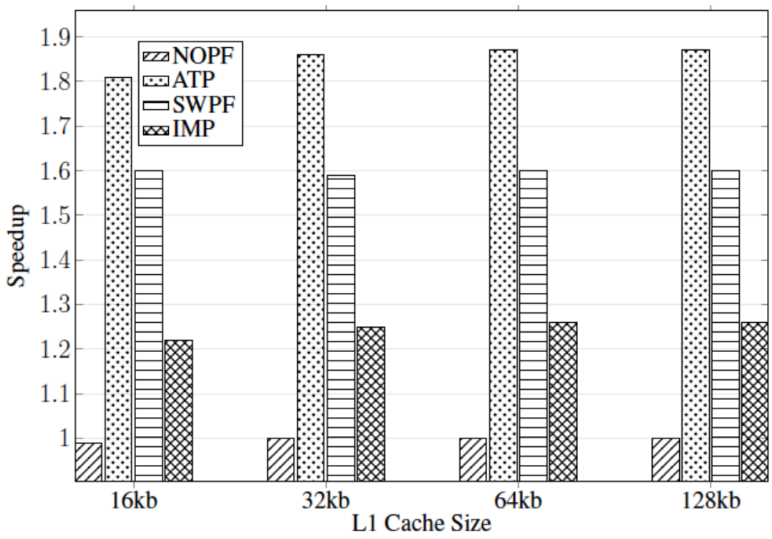


Figure 3.15: Effect of L1 Cache size in 4-core architecture. Baseline is no prefetching with 32KB L1 cache.

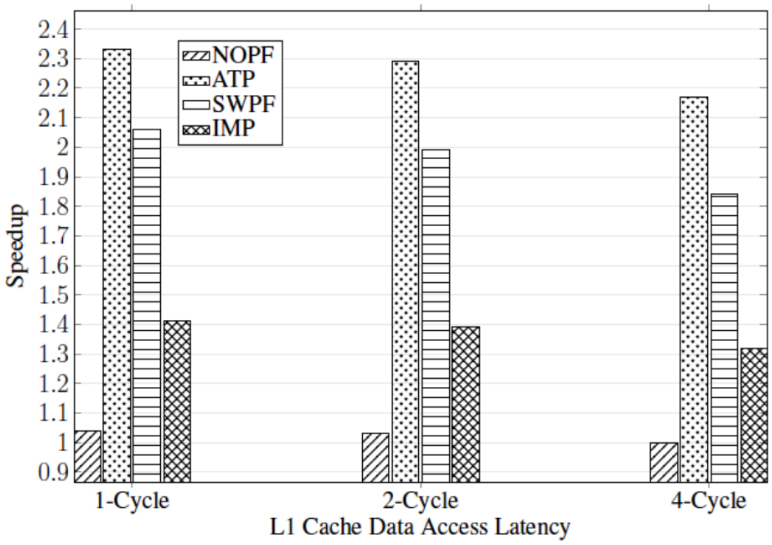


Figure 3.16: Effect of L1 Cache Data Access Latency in single-core architecture. Baseline for speedups is no prefetching with 4-cycle L1 cache latency.

Figure 3.18 shows the percentage of the dropped indirect prefetches compared to the total number of prefetches that were expected to be calculated (does not include the prefetches for index arrays). We observe that most of the dropped prefetches in benchmarks *is*, *cg*, *g500*, and *histo* are due to late prefetching of index values. In *g500*, some of the prefetches are dropped because their dependent

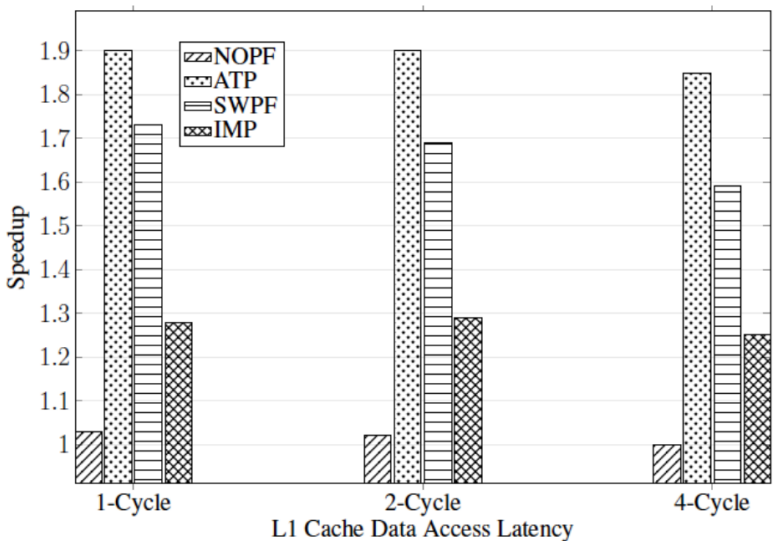


Figure 3.17: Effect of L1 Cache Data Access Latency in 4-core architecture. Baseline for speedups is no prefetching with 4-cycle L1 cache latency.

index values were not prefetched due to border conditions of the loop. Prefetch drops is most significant for *histo* followed by *g500*. Prefetch drops become more significant, especially as the number of cores in the system increases which impacts ATPs performance potential. As Figures 3.9 and 3.10 show, for 4 and 8-core configurations, SWPF outperforms ATP. In *pr* and *tc*, early prefetching of the index values causes indirect prefetch address calculations to fail. These benchmarks are issuing prefetches for the outer loop iteration, which means that we may see many memory accesses during the execution of the inner loop. This can cause the prefetched index values to be evicted from the cache.

Prefetch drops can be eliminated completely if the prefetch triggering is done when source data is placed in the cache. However, this requires significant changes in the ATP and is left as future work.

3.6 Related Work

Data prefetching is a well-known technique to help alleviate the memory wall problem [13] by increasing Memory-Level Parallelism (MLP) [14, 15]. Many

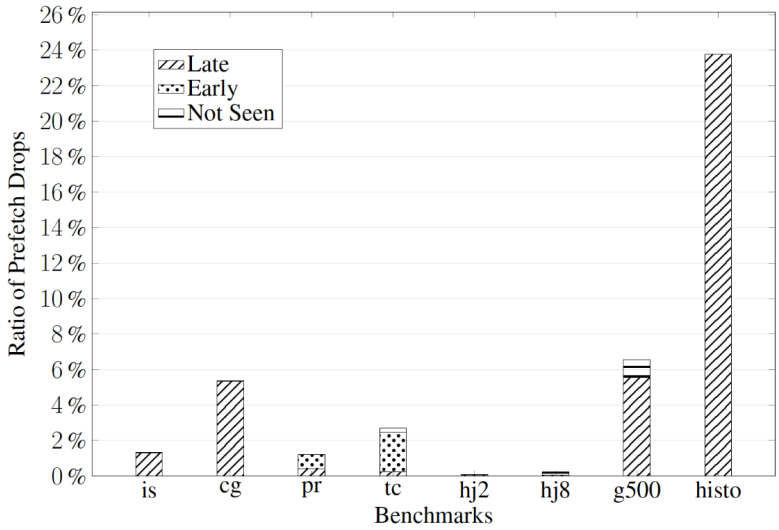


Figure 3.18: Ratio of dropped prefetches due to missing data required for indirect address calculation.

general-purpose microprocessors rely on data prefetching to improve performance for memory-intensive workloads. Most of the early prefetchers [16, 17, 18] were based on sequential prefetchers, which prefetch sequential memory blocks relying on the fact that many applications exhibit spatial locality. Although sequential prefetchers work effectively in many cases, applications with non-sequential data access patterns do not benefit from sequential prefetching. That motivated the research on more complex prefetchers that try to capture the non-sequential nature of these applications [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37]. Table 3.1 summarizes a variety of data access patterns and software and hardware prefetching methods targeting them. Prefetching techniques targeting pointer-based applications have been studied in [19, 20, 21, 33, 35, 37]. Indirect array references cannot be captured with those methods, however, since the desired addresses are computed, not contained in the memory as pointers. Guided Region Prefetching (GRP) [37] is a hardware prefetching scheme which uses compiler hints encoded in load instructions to regulate an aggressive hardware prefetching engine. GRP targets a broad range of behaviors from arrays and pointers to basic indirect

Table 3.11: Summary of Prefetching Methods for Basic Data Structures (+: full support; +/−: limited support; −: no support)

Basic Data Structure		Example Code	Pattern	Prefetching Method		
				Software	Hardware	ATP
Array	basic	$A[i]$	Stream	$A[i+D]$	stream [17]	+/−
	basic, pointer	$A[i]$, $A[i][constant]$	Stride	$A[i+D]$, $A[i+D][constant]$	stride [16], ghb [38], GRP [37]	+/−
	Indirect	$A[B[i]]$, $A[B[i][j]]$	Irregular	$A[B[i+D]]$, $A[B[i][j+D]]$ Ainsworth and Jones (SWPF) [2]	GRP, CDP [34], Markov [39], IMP [4]	+
	Indirect+pointer	$A[B[i]]{->}p$	Irregular	$A[B[i+D]]{->}p$ SWPF		+
Hash	basic	$A[func(i)]$, $A[func(B[i])]$	Irregular	push-pull [21], spaid [40], precomputation [31]	precomputation [32, 36]	+
	basic+pointer	$A[func(i)]{->}p$	Irregular	push-pull, spaid, precomputation	Precomputation	+
Linked Data Structures		$p = p{->}next$	Irregular	$p{->}next{->}next$, Luk and Mowry [41]	GRP, CDP, Markov, Pointer cache [33], jump pointer [35]	+/−
Mixed		Complex	Irregular/ Regular	Helper threads [30], precomputation	precomputation, Memory, streaming [24, 25, 26, 27, 28, 29]	+/−

arrays and recursive pointers. Indirect hints in GRP was limited to indirect array references of the form $A[B[i]]$ and assumes fixed array element sizes, thus missing significant performance potential.

More recently, Continuous Runahead Execution (CRE) [3] proposed a complex mechanism to dynamically identify address dependence chain of a load that is likely to create a cache miss. It can accurately prefetch data needed in the near future. However, CRE cannot provide effective prefetching for indirect accesses because indirect accesses create load miss chains, which prevent CRE to run sufficiently ahead. Most relevant to our work is the study by Yu [4] which proposed a hardware mechanism targeting indirect accesses. Although it can successfully find many regular (A[B[i]]), multi-way (A[B[i]] and C[B[i]]), and multi-level (A[B[C[i]]]) structures, it struggles to detect more complex structures and it cannot perform software specific optimizations (e.g., prefetching for the future iterations of outer loop instead of inner loop).

Software prefetching [42, 43, 44, 41, 40, 45] provides a way for programmers to insert prefetching instructions into a program targeting various simple and complex patterns. Manual insertion is flexible but requires significant programmer effort. Automatic insertion requires the compiler to recognize the access pattern. Ainsworth [2] developed an algorithm which automates the insertion of software

prefetches for indirect memory accesses into programs. Although this approach eliminates the requirement for the programmer effort, it cannot guarantee to insert the instructions in an optimized way for the specific architecture. Furthermore, significant instruction overhead may offset its benefits. On the other hand, software prefetching can target more complex patterns than hardware counterparts, especially if hardware budget is limited. In contrast to prior work, we proposed a hybrid software-hardware approach using the strengths of each for prefetching indirect memory accesses.

Finally, Lee et al. [45] studied the interaction between software and hardware prefetching and found that inserting software prefetching instructions in the presence of hardware prefetchers may hurt the overall prefetching performance due to the incorrect training of hardware prefetchers. ATP does not have this problem because prefetching is only initiated by hardware, not by software prefetch instructions; course-grain metadata instructions are used to guide the hardware prefetcher.

3.7 Conclusion and Future Work

We propose and implement the Array Tracking Prefetcher to have the benefits of both software and hardware prefetching for indirect memory accesses. ATP inserts prefetch metadata instructions outside the loop and use them to pass information to the hardware mechanism. The hardware mechanism uses this information to determine which indirect memory accesses to prefetch and when to do so. To increase the prefetch timeliness (and performance), ATP dynamically adjusts the distance. By using software hints, ATP avoids using an expensive hardware budget.

Our results show that ATP yields an average speedup of 2.17X, 1.85X, and 1.41X for single-core, 4-core, and 8-core architectures, respectively. ATP also out-

performs the state-of-the-art software-based (SWPF) and hardware-based (IMP) prefetching methods.

In future work, we plan to improve ATPs capacity by targeting diverse data structures. In this work, we showed that ATP can be extended to target linked-list traversals. However, our extension was based on a specific case where the number of nodes was known. In the future, we plan to improve our ATP software/hardware interface to enable prefetching for more complex data structure traversals. In addition, currently, ATP misses a significant opportunity by dropping prefetches when source data for address calculation is not present in the cache at the time of calculation. In future work, we plan to modify ATP so that prefetch address calculations can be triggered by cache fills from source data due to prefetched index array.

List of References

[1] M. Cavus, R. Sendag, and J. J. Yi, "Array tracking prefetcher for indirect accesses," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 132–139.

[2] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 305–317.

[3] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 61.

[4] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 178–190.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[7] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 44–55.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[9] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on computing*, vol. 14, no. 1, pp. 210–223, 1985.

[10] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 362–373.

[11] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[12] I. R. Group *et al.*, "Parboil benchmark suite," 2007.

[13] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[14] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 301–313.

[15] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 222–233, 2005.

[16] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE, 1991, pp. 176–186.

[17] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 364–373.

[18] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

[19] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 7–17.

[20] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 115–126, 1998.

[21] C.-L. Yang and A. R. Lebeck, "Push vs. pull: Data movement for linked data structures," in *Proceedings of the 14th international conference on Supercomputing*. ACM, 2000, pp. 176–186.

[22] P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan, "Exploring the limits of prefetching," *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 127–144, 2005.

[23] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 126–140, 2004.

[24] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 69–80.

[25] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 79–90.

[26] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 1–10.

[27] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 99–108.

[28] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 266–277, 2007.

[29] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 252–263.

[30] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings 29th Annual International Symposium on Computer Architecture.* IEEE, 2002, pp. 171–182.

[31] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings 28th Annual International Symposium on Computer Architecture.* IEEE, 2001, pp. 40–51.

[32] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 2001, pp. 306–317.

[33] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society Press, 2002, pp. 62–73.

[34] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *ACM SIGPLAN Notices*, vol. 37, no. 10. ACM, 2002, pp. 279–290.

[35] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 111–121.

[36] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 2–13, 2001.

[37] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE, 2003, pp. 388–398.

[38] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, Feb 2004, pp. 96–96.

[39] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 252–263. [Online]. Available: http://doi.acm.org/10.1145/264107.264207

[40] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "Spaid: Software prefetching in pointer-and call-intensive environments," in *Proceedings of the 28th annual international symposium on Microarchitecture.* IEEE Computer Society Press, 1995, pp. 231–236.

[41] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5. ACM, 1996, pp. 222–233.

[42] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 40–52.

[43] T. C. Mowry, "Tolerating latency through software-controlled data prefetching," Ph.D. dissertation, to the Department of Electrical Engineering.Stanford University, 1994.

[44] Y. Wu, M. Serrano, R. Krishnaiyer, W. Li, and J. Fang, "Value-profile guided stride prefetching for irregular code," in *International Conference on Compiler Construction*. Springer, 2002, pp. 307–324.

[45] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesnt, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, p. 2, 2012.

CHAPTER 4

Informed Pre-Execution for In-Memory Database Applications

Mustafa Cavus[1], Resit Sendag[2], Mohammed Shatnawi[3]

[1,2,3]Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI02882.

4.1 Abstract

Pointer-intensive data structures are commonly used in database applications. Traversals on these data structures mostly cause memory stalls due to their dependent pointer references. Improving memory-level parallelism by accessing the memory simultaneously for separate lookups is beneficial for such data structures. Existing techniques focus on improving their performance by creating overlapping memory accesses for distinct lookups. Although data prefetching is very beneficial for such structures, it is not enough alone to maximize their performance on modern CPUs.

In this work, we propose Node Tracker (NT), a software supported hardware prefetching mechanism which is tightly integrated with CPU. Additionally, it can use the extracted knowledge from the prefetched data to inform out-of-order CPUs about the future matching nodes and conditional branch targets. In our evaluations, NT achieved up to 19x speedup over no-prefetching baseline.

4.2 Introduction

Pointer-intensive data structures, such as linked-lists and trees are used by many in-memory database applications. These applications have unpredictable access patterns due to frequent pointer chasing and result in leaving the CPU idle due to long memory latencies. Although modern CPUs utilize memory-level parallelism (MLP), the benefit of MLP comes from the number of independent in-flight accesses. In pointer chasing lookups, accessing the next hop requires data from previous pointers, which prevents the CPU to service these accesses in parallel. Moreover, many database operations consist of multiple lookups that can be serviced in parallel but the CPU has a fixed instruction window size which limits the number of simultaneous lookups.

Data prefetching is intended to hide memory access latencies in single-core

and multi-core systems which effectively reduces the gap between memory access time and processor frequency. Previous software-based solutions [1, 2] exploit inter-lookup parallelism to overlap memory access latencies. However, they require to re-design the algorithm and still they have limited benefits due to dependencies and hardware limitations. Helper threads [3] can create a separate thread to issue prefetches. However, they eventually tend to stall and struggle to be ahead of the main thread due to load-miss chains created by pointer-intensive applications. Ainsworth [4] proposed a system with programmable cores to issue prefetches for future accesses. Although it provides an ideal solution for prefetching, the pre-executions they perform can only help reducing demand cache misses.

In this work, we propose Node Tracker (NT), a software-assisted hardware prefetching mechanism, that is highly integrated with CPU pipeline. NT relies on programmer/compiler to configure the hardware. NT focuses on pre-executing multiple future lookup operations on pointer-intensive data structures asynchronously using simple in-order programmable cores. In addition to prefetching, NT assists CPU execution by eliminating unnecessary node visits and providing the future branch targets to the CPU pipeline. NT is designed to be integrated with another prefetcher, ATP [5], which handles prefetching for sequential and indirect accesses. ATP also provides necessary information to NT to be able to start pre-execution of future lookups.

4.3 Related Work and Motivation

Code Snippet 4.1 illustrates an example of a simple probe hash-table. The algorithm consists of two loops, an outer loop that performs a lookup of a key value in a linked list, and an inner loop that compares the key with the currently visited node. Once a matching node is found, it moves on to the next key lookup.

The inner loop visits a single node in each iteration. Due to its pointer chas-

Code Snippet 4.1: A simple example code to demonstrate probe hash-table algorithm.

```
[CONFIGURATION INSTRUCTIONS]
[outer loop]
for (i = 0; i < NUM_TUPLES; i++) {
    int key = keys[i];
    int idx = HASH(key);
    node *n = ht[idx];
    while (n) { [inner loop]
        if(key == n->keys){
            matches++;
            break;
        }
        n = n->next;
    }
}
```

ing behavior, a lookup on a linked-list creates dependent memory accesses where it needs to access the nodes of a linked-list sequentially. Other pointer-intensive workloads like binary tree search (BST) also has similar dependent memory accesses where the performance of the application depends on the number of nodes to be traversed for a single lookup.

Several proposed prefetching techniques are able to cover prefetching pointer-chasing accesses across distinct lookups. Kocberber [2] presented a software-based method called AMAC that exploits parallel lookups for pointer-intensive database applications. AMAC proposes a way to implement the algorithm to be able to serve multiple key lookups asynchronously to hide long memory access latencies by improving MLP. Although this method aims to parallelize node accesses across different lookups, it needs prefetches to be timely accurate to achieve the best performance. Late prefetches may cause stalls, while early prefetches may lead having a nodes pointer in the cache but unable to create the next pointers request on time.

Amir [6] proposed a robust technique to prefetch jump pointers to tolerate linked lists access time named JPP (Jump Pointer Prefetching). JPP explicitly stores jump pointers to nodes located a number of hops away. It can be beneficial for long pointer chains which are not commonly used by database applications we have.

Helper threads [3] can run different context on a separate thread to create load accesses ahead of the main thread without any need of extra hardware. However, the additional thread which runs on a high-performance core could consume significant amounts of energy. They are also unable to calculate prefetches ahead of demand execution where load-miss chains are common.

Ainsworth [7] proposed a design with programmable cores to compute prefetch addresses. They can accurately prefetch irregular access patterns without modifying the original code. However, they do not provide any support for CPU execution except hiding memory access latencies by prefetching.

We designed a prefetching/pre-execution technique, Node Tracker (NT), targeting pointer-intensive data structures with multiple lookup operations which is common in in-database applications. In addition to prior work, NT uses the knowledge of pre-executed lookups to provide out-of-order CPUs further support.

4.4 Node Tracker

Figure 4.1 represents a block diagram of the whole system with node tracker. Node Tracker is a programmable unit that is configured using special instructions inserted by programmer/compiler before the outer loop as it is marked in Code Snipped 4.1. Using the information passed from the software, NT can prefetch node addresses of future lookups. It basically traverses over multiple future lookups ahead of demand execution. These special instructions are recognized by the CPU pipeline and inserted into a dedicated FIFO instruction queue called Configura-

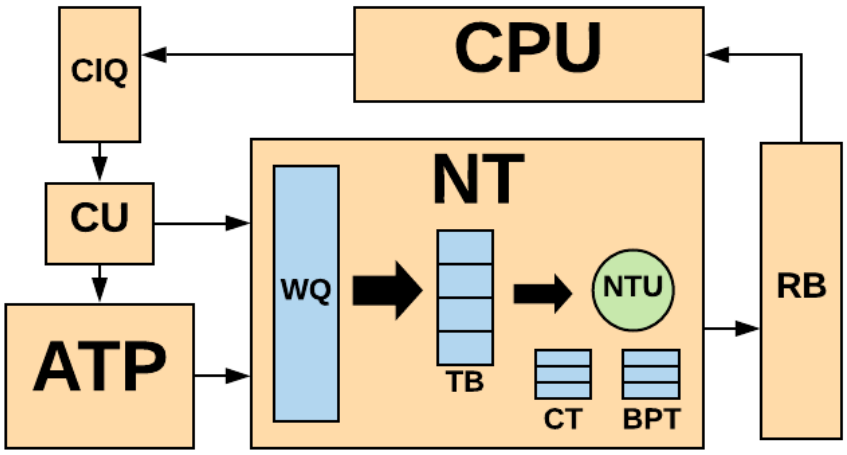


Figure 4.1: An overview of Node Tracker.

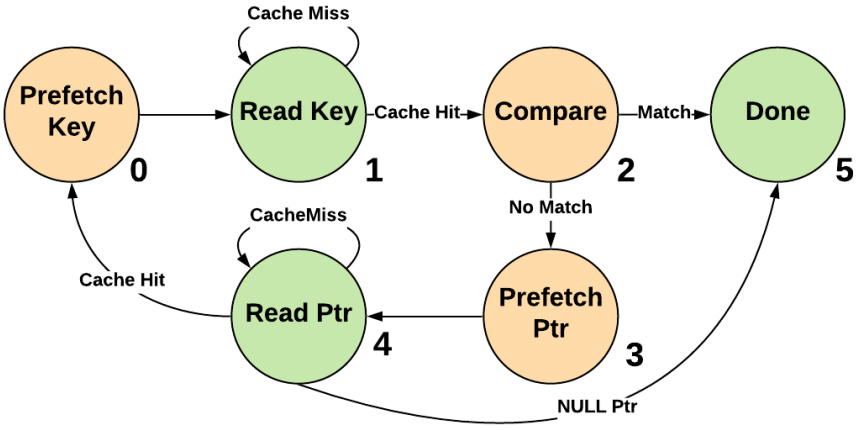


Figure 4.2: State diagram of lookup execution in NTP.

tion Instruction Queue (CIQ). The instructions are then executed by a simple unit called Configuration Unit (CU) that is designed to initialize some dedicated registers that keep useful information about the software. To be able to start prefetching for a future lookup, NT needs to know which head node address to start the lookup from and which key-value it is searching for. These values can be provided by ATP which is designed to prefetch indirect memory accesses. ATP can compute the addresses of future sequential or indirect array accesses (for example, A[B[i]] uses B[i] to indirectly access array A). In the lookup example code (Code Snipped 4.1), ATP can prefetch the elements of "*keys*" array and "*ht*" ar-

ray (which holds the head node pointers) by using the PC of the load instructions passed from the software. When ATP calculates the prefetch address of a head node, it inserts an entry (consists of the head node pointer and key-value) into a FIFO queue called Wait Queue (WQ) which is a part of Node Tracking Prefetcher (NTP) as shown in figure 4.1.

NT keeps the in-flight lookup operations in another buffer called Task Buffer (TB). Whenever an available TB entry is detected, a new lookup task from the head of WQ is inserted into TB. The number of TB entries determines how many lookup tasks can be carried simultaneously. The tasks in TB are executed once at a time in round-robin fashion. NTP switches to the next task whenever a cache miss occurs for the current task or it completes the lookup. Once all task entries are visited, NTP starts over executing from the first entry. A simple processing unit, Node Tracking Unit (NTU), is shared amongst the TB entries which handles the address computations and comparisons.

The lookup operations are executed by a simple state machine by using the pre-configured settings of NT. Figure 4.2 demonstrates a simple execution flow of a TB entry. NTP switches to the next entry in two cases; either when there is a cache miss occurs (states 1 and 4), or the task is completed (state 5). When NTP switches to a new entry and activates it, it continues executing from the previous state. To remember the latest state, each TB entry has a state field. If no task is assigned (either previous task of the entry is completed or it is activated for the first time), then it fetches an entry from the head of WQ. Each TB entry keeps lookup ID, node pointer, and key values that are carried from WQ.

To perform the comparison, the comparison operations and corresponding actions should be passed to the hardware. Comparison operations may refer to any of the comparison parameters like " == " and " < " as well as "*nocomparison*"

which means that the corresponding actions will be performed without any comparison. And there are two types of actions to be performed in NT; "$EXIT$" which finishes the lookup if the comparison is true and "$NEXT$" which reads and sets the next node pointer and continues. These comparison operations and corresponding actions are stored in a small table called Comparison Table (CT). CT entries also have a field to keep an offset value to be used with "$NEXT$" action and refer to the offset from the node address to read the pointer of the next node. Each comparison execution starts with the first entry and for each false comparison, it moves to the next entry. In Code Snippet 4.1, there is one comparison operation ("$key == n- > key$") and it corresponding operations is "$EXIT$" since the matching node is found, if the comparison is true. If the first comparison is false, it needs to move to the next node. In this case, the second entry should be inserted as "$nocomparison$" with the action of "$NEXT$" and the offset should be set.

4.4.1 Result Buffer and Node-Update

Like in the example shown in Code Snippet 4.1, a lookup operation consists of several node visits to compare the keys till the matching node is found. Since NT already pre-executes these node visits, the CPU pipeline does not need to revisit the non-matching nodes again in most cases. Thus, a direct-mapped buffer called Result Buffer (RB) stores the matching node addresses for the completed lookups. CPU pipeline can read the node addresses from RB and instead of starting the inner loop from the head node, it can start from the matching node (inner loop can be executed for one iteration in this case). To achieve this, using the special instructions, we need to inform the hardware about the pc (hold pc) of the instruction which writes the pointer address into a register.

CPU uses a lookup ID to read the matching node addresses from RB. Lookup

IDs are assigned to the instructions by the lookup counter in the fetch stage. The lookup counter keeps the lookup ID of the last fetched instruction. Its value is increased each time a new lookup (outer loop iteration) begins. So, every instruction within the same outer loop iteration has the same unique lookup ID. If any of the instructions are discarded due to a branch misprediction, the counter value is restored with the value of the latest valid instruction's lookup ID. For the future lookups, IDs are generated by ATP and inserted into WQ along with the key and the pointer to the head node. Later these IDs are used to index RB. Whenever a new task is assigned to a TB entry, a result entry in RB is allocated for the corresponding lookup. Each entry of RB keeps a ready flag which is set to false initially and once the lookup is completed and the matching node address is written to its pointer field, it is set to true which means that it is ready to be read by the CPU.

CPU does not let the instructions (hold instructions) with hold pc to issue directly. First, it accesses to the RB using the lookup ID of the instruction. If the corresponding entry exists and the ready flag is set to true, it reads the matching node address and replaces the value of the result of the instruction with the matching node address. If the RB entry is available but its value is not ready (still processing in TB), the hold instruction is not issued till NT finishes the lookup writes the matching node address into to RB entry. In the later sections of this chapter, we will call this process as node-update (NU).

4.4.2 Branch Buffer and Branch-Fix

By prefetching with NT, we observe that we can almost eliminate all of the demand cache misses in the applications we evaluated. However, even though node-update reduces the number of branches per lookup significantly, we still observe a bottleneck due to branch mispredictions. Since the number of branches per lookup is very low with node-update (due to the single iteration in the inner loop), the

actual branch outcomes can be extracted from the pre-executed lookups to set the branch targets in CPU pipeline.

To achieve this, branch patterns for different scenarios should be generated as bit vectors and passed to the hardware using special instructions. Each branch pattern is stored in an entry in Branch Pattern Table (BPT) and each entry is mapped to a unique scenario (node matched, node not matched, etc.). In Code Snipped 4.1, there are two branches per inner loop iteration ("$while(n)$" and "$if(key == n- > key)$"). If the node "n" is a matching node, both branches are expected to be taken (represented as "11"). Whenever NT completes a lookup and writes output to RB, it also writes the corresponding branch buffer. In our applications, the number of unique scenarios we observe for a single inner loop iteration does not exceed 4 which the maximum number of BPT entries we need. In the later sections of this chapter, we will address this method as branch-fix (BFX).

4.5 Methodology

We implemented NT on gem5 simulator and evaluated results using x86 out-of-order CPU model in System Emulation mode. Table 4.1 shows our configurations for the simulator, NT, and ATP.

For hash-join and binary search tree workloads (including the baseline algorithms and AMAC implementations), we used the implementation of Kocberber et al. [2]. The performance of NT is evaluated using probe has-table algorithm with 4 nodes per bucket (PHT-B4), 8 nodes per bucket (PHT-B8), and with a random node distribution (PHT-RND). Additionally, also a binary search tree (BST) algorithm is used for the experiments.

Table 4.1: Simulation, NT, and ATP Configurations

ISA	64-bit x86
Architecture	4-issue, out-of-order, 20-stage, 2GHz
LQ/SQ Entries	96/64
IQ/ROB Entries	128/256
Br. Pred.	Tournament BP
L1 Cache	64 KB, 24 (single)/16 (multi-core) MSHRs
L2 Cache	256 KB, 24 (single)/16 (multi-core) MSHRs
LLC Cache	1 MB per core, 48 MSHRs
Memory	8 GB DDR3, 256-bit System Bus, 1 (single)/8 (multi) Mem. Ctrl.
Node Tracker	48-entry WQ, 16-entry TB, 128-entry RB, 128-entry BB
Array Tracker	4-entry IQ, 4-entry AT, 4-entry RT, 2-entry OT, 4-entry Pf. B.

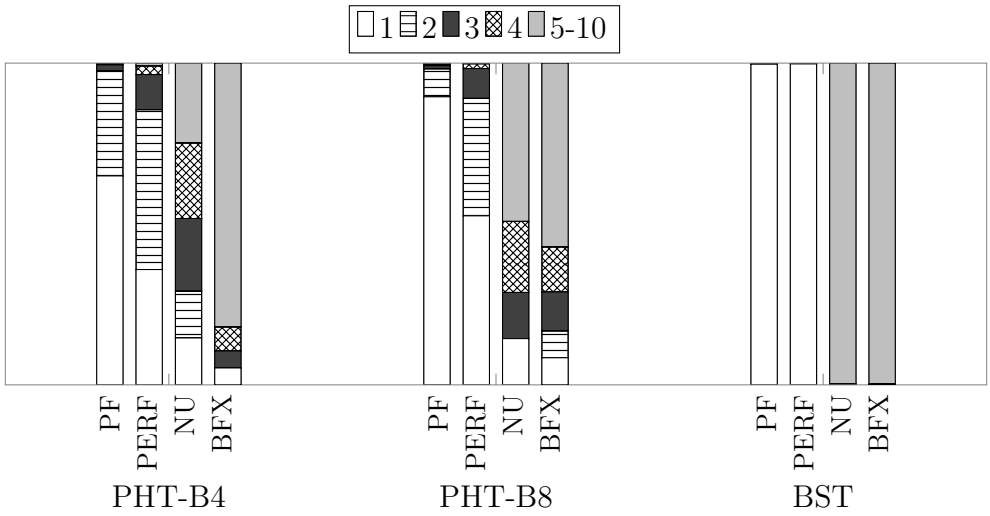


Figure 4.3: Rate of number of lookups run in CPU instruction window simultaneously.

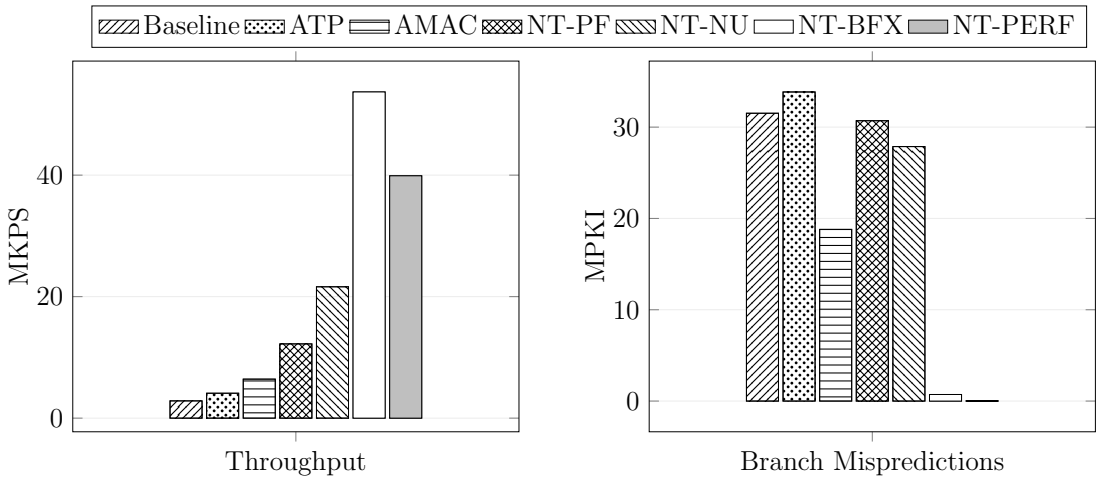


Figure 4.4: Throughput and branch misprediction rate comparison of PHT-B4.

4.6 Results

We compared our results with no-prefetching baseline, ATP, and AMAC. First, we discussed how NT performs by prefetching only, and then we discussed the effect of node-update and branch-fix. We used the number of million keys per second (MKPS) as a metric to measure the throughput of different methods.

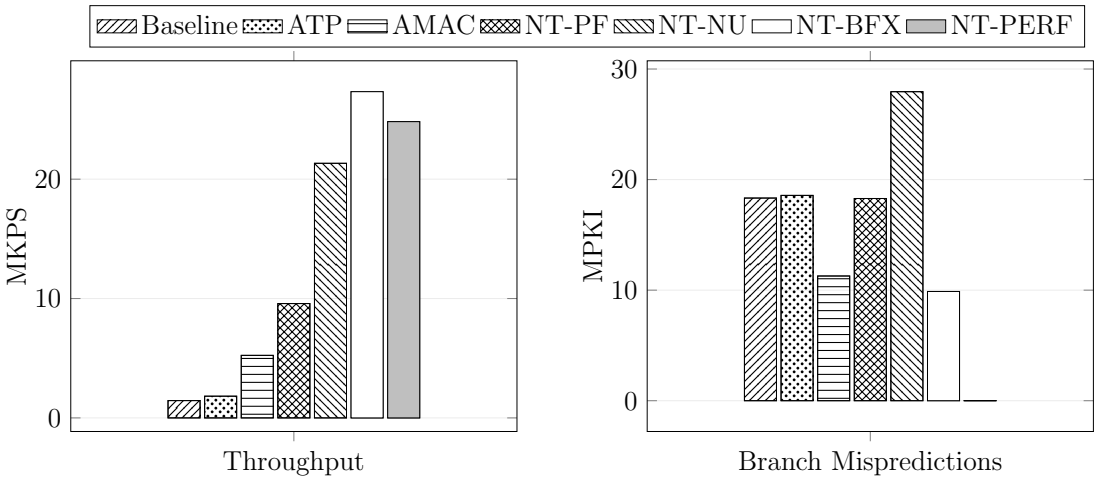


Figure 4.5: Throughput and branch misprediction rate comparison of PHT-B8.

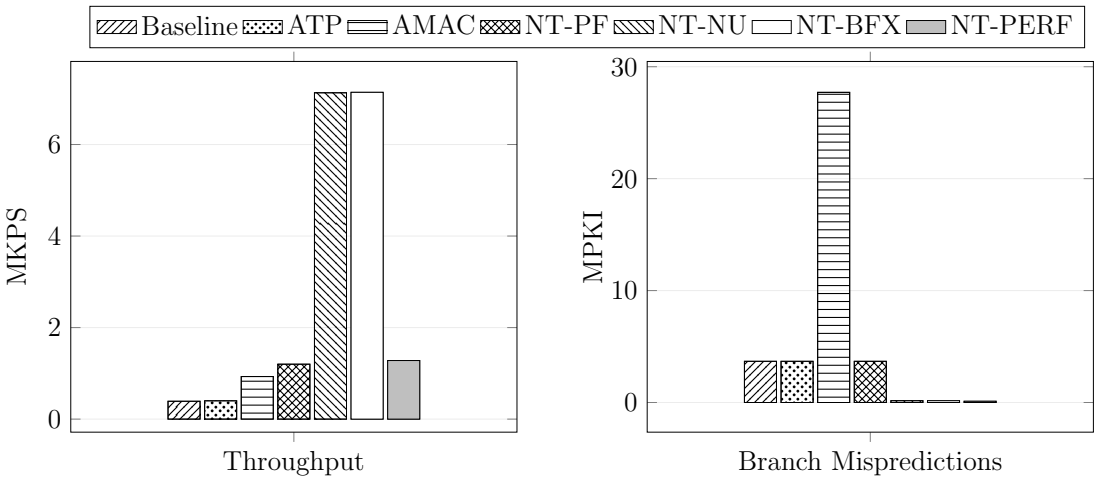


Figure 4.6: Throughput and branch misprediction rate comparison of BST.

4.6.1 NT Prefetcher Performance

Figures 4.4 to 4.6 shows the throughput (million keys per second) for each method we evaluated on a single out-of-order core. These results indicate that NT with prefetching has a significant benefit in all benchmarks over baseline with no-prefetching.

NT with prefetching (NT-PF) has 4.3x, 6.6x, and 3.1x speedups for PHT-B4, PHT-B8, and BST respectively. PHT-B8 has longer linked-list traversals compared to PHT-B4 which reduces its cache locality and baseline throughput. Thus, it

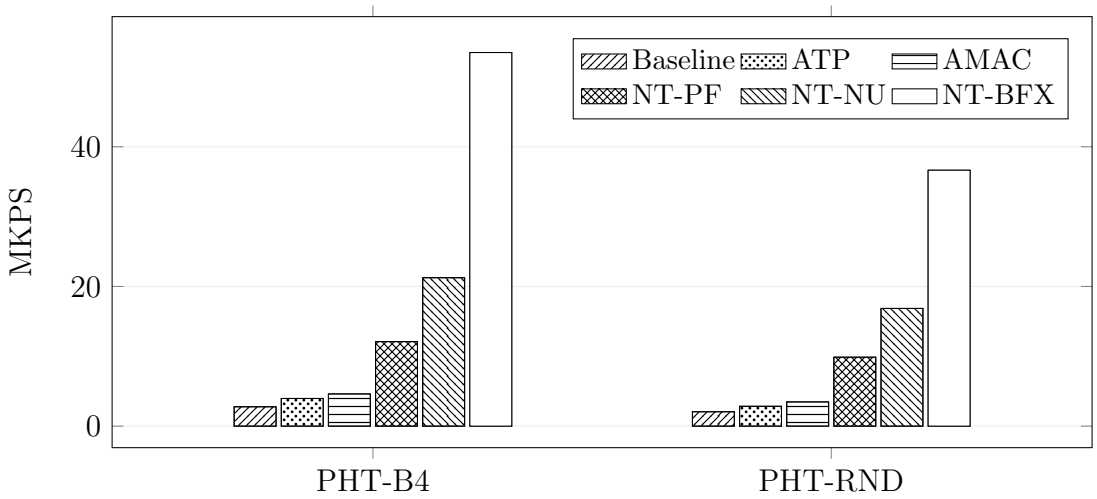


Figure 4.7: Comparison between fixed number of nodes per bucket and randomly distributed nodes across the buckets.

benefits from prefetching better than PHT-B4 and has a higher speedup with NT-PF.

Even though BST benefits quite well from NT-PF, it has a lower speedup compared to PHT-B4 and PHT-B8. BST has very long traversals compared to PHT-B4 and PHT-B8. Due to its long dependency trees, CPU pipeline mostly processes a single lookup at a time (see Figure 4.3). So, even though NT-PF improves its performance by eliminating cache misses, its performance is limited by the instruction window.

NT-PF has 4.3x, 6.6x, and 3.1x speedups for PHT-B4, PHT-B8, and BST respectively which is better than ATP and AMAC in all benchmarks. ATP can benefit from prefetching sequential and indirect array accesses. It has low (on PHT-B4 and PHT-B8) or no speedups (on BST) over the baseline which means that most of the speedup of NT-PF comes from prefetching the nodes.

AMAC has 2.3x, 3.6x, and 2.4x speedups over no-prefetching baseline on PHT-B4, PHT-B8, and BST respectively. But its performance is also very limited due to the dependent loads causing stalls. NT-PF has almost two times higher

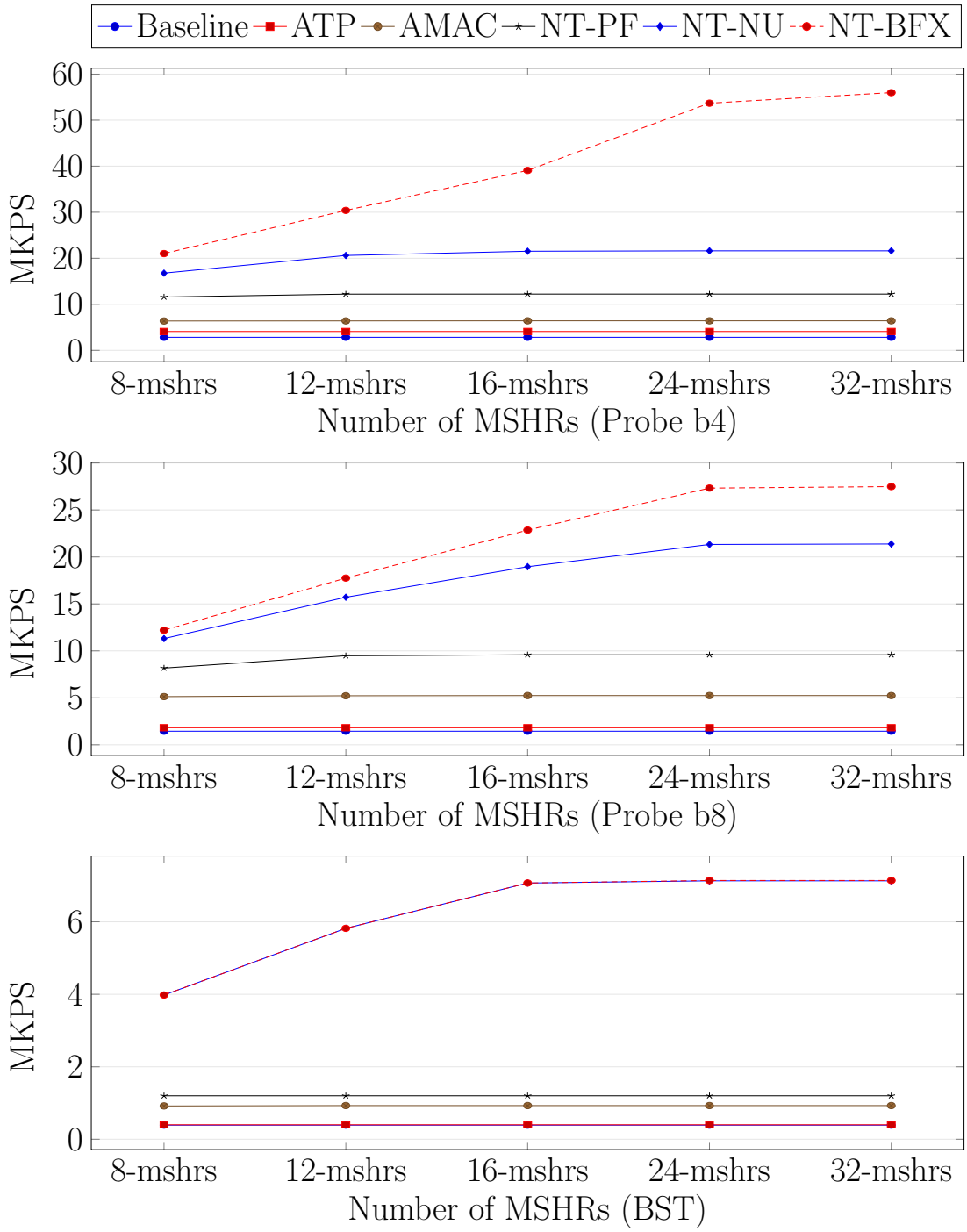


Figure 4.8: Impact of number of MSHRs in L1 cache for all benchmarks.

throughput over PHT-B4 and PHT-B8. It is also 1.3 times faster on BST.

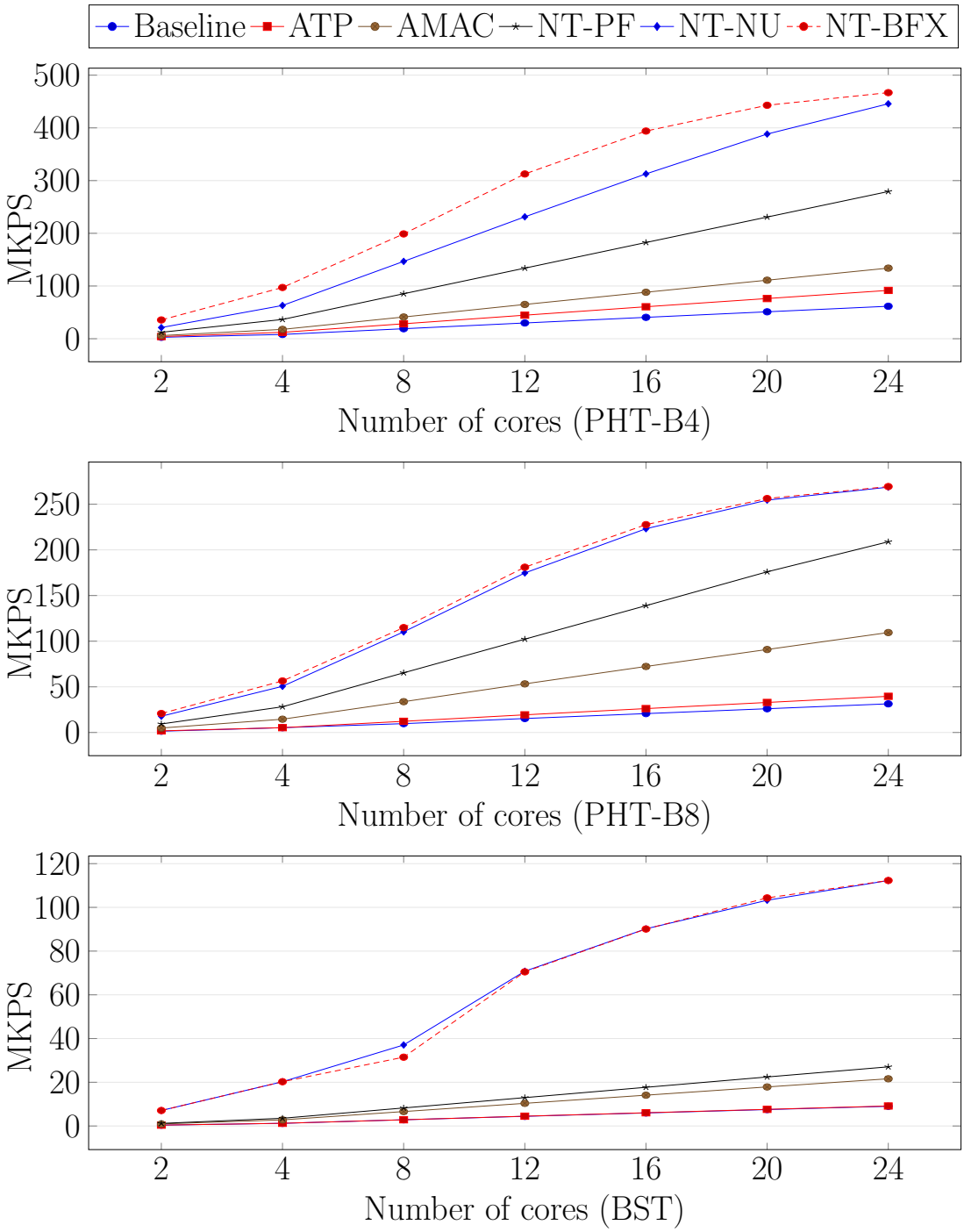


Figure 4.9: Scalability comparison of all methods with increasing number of cores.

4.6.2 Impact of Node-Update and Branch-Fix

Figure 4.3 shows the number of lookup rate processing simultaneously in the CPU instruction window. Using node-update (NT-NU) improves the number of

lookup rate for all benchmarks significantly by shortening the dependency chains.

Using node-update (NT-NU) improves the performance significantly for all benchmarks (refer to Figures 4.4 to 4.6). For PHT-B4 and PHT-B8, NT-NU has 1.8x and 2.2x better performance respectively compared to NT-PF (refer to Figures 4.4 to 4.6). The effect of node-update of BST even higher than other benchmarks. Because the number of lookup rate of BST was 1 all the time with NU-PF (refer to figure 4.3). With node-update, CPU can process 5 or more lookups simultaneously.

To study the effect of branch mispredictions, we evaluated NT-PF using a perfect branch predictor (NT-PERF) to see the performance benefit when we eliminate the cache misses and branch mispredictions at the same time. NT-PERF improves the throughput 3.3x and 2.6x over NT-PF for PHT-B4 and PHT-B8 respectively. By eliminating all the branch mispredictions, NT-PERF also increases the number of lookup rate in instruction window as it is seen in figure 4.3. BST uses conditional move instructions which perform better than branches due to its tree structure and it leads to having long dependency chains instead of having high branch misprediction rates. However, it prevents BST benefiting from NT-PERF. Also, AMAC does not use conditional move instructions for BST and it's misprediction rate is significantly higher than the baseline (refer to figure 4.6).

Figures 4.4 to 4.6 also show the throughput and branch misprediction rate of NT-PERF which is NT-PF with perfect branch prediction. For PHT-B4 and PHT-B8., eliminating branch mispredictions have a big impact. Although NT-NU improves the number of lookups rate in CPU instruction window better than NT-PERF, NT-PF has very high branch misprediction rates which cause many of those iterations are actually flushed.

Using NT-BFX eliminates most of the branch mispredictions and it improves

the overall performance even better then NT-PERF since it has the advantage of using node-update. NT-BFX improved the throughput of NT-NU by 2.5x and 1.3x for PHT-B4 and PHT-B8 respectively. Due to longer traversals of PHT-B8, some of the NTUs are being late to save the branch outcomes to BB. figure 4.5 that NT-BFX has an MPKI of 10 which means that it is not able to cover all the branches. We could increase the prefetch distance to compute branch outcomes more ahead of the execution but it requires us to increase buffer sizes and leads to increase cache misses due to early prefetches which might decrease the overall throughput. So, we decided to keep the distance as 32 for our evaluations to have the best overall performance on all benchmarks. On BST, we do not see any benefit of branch-fix since it already has almost zero MPKI with NT-NU (refer to figure 4.6).

4.6.3 Impact of Node Distribution in Hash-Join Probe

We also examined the effect of random distribution of nodes by simulating another benchmark, PHT-RND, which has a random number of nodes per each bucket (distributed using Zipfian with factor 0.5) and the total number of nodes is equal to the total number of nodes in PHT-B4. Figure 4.7 shows a comparison between PHT-B4 and PHT-RND. The throughput of PHT-RND is lower compared to PHT-B4 in general but almost all methods we used have similar speedup values in both benchmarks. Only NT-BFX tend to have a slightly better speedup on PHT-B4 compared to PHT-RND. The reason for that is NT may not produce the outcomes of the longer lookups (which have a higher number of nodes) on time but still, it can manage to complete most of them.

4.6.4 Impact of Number of MSHRs

We simulated all methods with different number of MSHRs as seen in Figure 4.8. We observe that NT-BFX benefit of the increased number of MSHRs for all benchmarks. NT-NU also benefit of the higher number of MSHRs on PHT-B8 and BST but it does not impact the performance of NT-NU on PHT-B4 since its performance is limited due to branch mispredictions. Depending or experiments, we decided to use 24 MSHRs for single and 2-core architectures, 16 MSHRs for architectures with 4 or more number of cores.

4.6.5 Multicore Scalability

We also observed how NT and other methods we tested scales as we increase the number of cores. Figure 4.9 shows how throughput scales as we increase the number of cores using each method and benchmark. Up to 12 cores, NT-NU and NT-BFX scale perfectly on every benchmark. After 12 cores scaling for NT-NU and NT-BFX on PHT-B8 and BST starts to slow down but still throughput is increasing well up to 24 cores. In PHT-B4, NT-NU scales very well up to 24 cores but NT-BFX starts to lose its advantage after 16 cores but still performs better than NT-NU.

4.7 Additional Discussions

We proposed NT as a prefetcher unit which pre-executes future lookup operations on pointer-chasing traversals but the significant benefit of NT relies on its tight integration with CPU. Another approach to address the typical applications would be designing an accelerator which performs the lookups independently from the CPU as proposed by [8]. Although such an accelerator could do the job as efficient as NT, this would require the accelerator to be able to perform everything CPU does. NT instead, still lets the CPU to execute the application so it does not

need to support complex operation as an accelerator needs to do.

4.8 Conclusion

We proposed Node Tracker as a prefetcher unit which pre-executes future lookup operations on pointer-chasing traversals but the significant benefit of NT relies on its tight integration with CPU.

NT with prefetching only achieved up to 6.6x speedup. NT with informing CPU about the matching node pointer achieved a maximum speedup of 18x and when NT also provides branch outcomes by using the knowledge it received from the software and prefetched data, it can achieve up to 19x speedup in our evaluations over no-prefetching baseline.

List of References

[1] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 3, p. 17, 2007.

[2] O. Kocberber, B. Falsafi, and B. Grot, "Asynchronous memory access chaining," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, 2015.

[3] Changhee Jung, Daeseob Lim, Jaejin Lee, and Y. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, April 2006, pp. 10 pp.–.

[4] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," *SIGPLAN Not.*, vol. 53, no. 2, pp. 578–592, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3296957.3173189

[5] M. Cavus, R. Sendag, and J. J. Yi, "Array tracking prefetcher for indirect accesses," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 132–139.

[6] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 111–121.

[7] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 305–317.

[8] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540748

CHAPTER 5

Conclusion

Hardware prefetching has been a subject of academic research and industrial development for over 40 years. Nevertheless, because of the scaling trends that continue to widen the gap between processor performance and memory access latency, the importance of hardware prefetching and the need to hide memory system latency has only grownfurther innovation remains critical.

Sequential prefetchers are useful for many workloads but it is critical for them to issue prefetches timely. Late prefetches cannot hide the latency sufficiently. In this case, when the prefetched data is accessed by the demand execution, the data has not arrived at cache on time so it creates a cache miss. If the prefetch is issued too early, the prefetched data might be replaced with other data before it is accessed by the demand execution. In this case, demand execution creates another cache miss for the previously prefetched data since it is not in the cache anymore. Also, some applications do not benefit of sequential prefetchers and they can even decrease their performance by polluting caches with unnecessary data and wasting bandwidth resources. In chapter 2 we proposed Sequential Prefether with Adaptive Distance (SPAD) which monitors either the prefetches are being late or early and adjusts distance dynamically. It also monitors if prefetching is useful or not and it turns it off when it is not beneficial. It achieves a 20% speedup on average on the benchmarks we evaluated and outperforms most recent sequential prefetching methods.

Although sequential prefetching is beneficial for a wide variety of workloads, many other applications create irregular memory access patterns which cannot be captured by sequential prefetchers and their performance is critical for certain

fields. An important portion of these applications involves indirect memory accesses which are widely used in data structures like graphs, sparse matrices, etc. Software prefetching is very useful for indirect memory accesses but the insertion of software prefetching instructions increases the number of instructions to execute which might create overhead in some cases. Also, software prefetching requires programmer knowledge and effort to tweak to achieve the best performance based on the target microarchitecture. Hardware mechanisms developed to capture indirect memory accesses but without knowledge from the software, they are unable to detect complex indirect access behaviors. We proposed Array Tracking Prefetcher (ATP) in chapter 3 which combines the strengths of software and hardware methods to capture a variety of indirect access behaviors and it outperforms software and hardware-based methods width a speedup of 2.17 and 1.84 in single-core and 4-core architectures respectively.

Pointer-chasing memory accesses are also hard-to-predict by pure hardware mechanisms. Also, dependent access chains make it very hard for both hardware and software mechanisms to keep ahead of the demand execution. Fortunately, many database applications involve multiple lookup operations on linked-data structures. This brings the opportunity to benefit create cache misses of different lookup operations in parallel. Software methods require to modify the algorithm to be able to benefit from inter-lookup parallelism and it increases the number of instructions per lookup significantly which limits their potential. In section 4, we proposed a software supported hardware mechanism, Node Tracker (NT), which is designed to pre-execute future lookup operations. It prefetches the data of the future lookups, but it also helps demand execution in other ways to boost the performance even further. Since NT pre-executes and finds the matching node of the linked data structure, it informs demand execution with the matching node

pointer so demand execution does not need to iterate all the nodes again to find it. Branch-mispredictions are also a very important cause which limits the performance of the workloads we evaluated. NT records the expected branch outcomes in a buffer to provide it to the branch-predictor later. Using all these features, NT can be considered as a helper unit to the CPU which not only improves the performance by reducing cache misses as a prefetcher, it also helps CPU execution by providing all the knowledge it collects by pre-executing future lookup operations. NT achieves up to 19x speedup by using its all features to boost CPU execution performance.

5.1 Future Work

In future work, we will work on developing hardware-aware compilers. Although we have introduced special instructions to inform hardware mechanisms, compilers still optimize the programs without any knowledge of the hardware we designed. In this case, the hardware mechanism is required to support a wide variety of assembly code structures which brings extra hardware overhead and limitations. If the compiler knows how we designed the hardware and what kind of code structures it will benefit more, it can use the optimizations carefully and even modify the structure of the code to help our mechanisms to work more efficiently.

BIBLIOGRAPHY

"Standard performance evaluation corporation cpu2006 benchmark suite." [Online]. Available: http://www.spec.org/cpu2006/

"Dpc-1.1stjilpdataprefetchingchampionship." 2009. [Online]. Available: http://www.jilp.org/dpc/

"Dpc-2. second data prefetching championship." 2015. [Online]. Available: http://comparch-conf.gatech.edu/dpc2/

Ahmad, M., Hijaz, F., Shi, Q., and Khan, O., "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 44–55.

Ainsworth, S. and Jones, T. M., "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 305–317.

Ainsworth, S. and Jones, T. M., "An event-triggered programmable prefetcher for irregular workloads," *SIGPLAN Not.*, vol. 53, no. 2, pp. 578–592, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3296957.3173189

Annavaram, M., Patel, J. M., and Davidson, E. S., "Data prefetching by dependence graph precomputation," in *Proceedings 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 52–61.

Baer, J.-L. and Chen, T.-F., "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 176–186. [Online]. Available: http://doi.acm.org/10.1145/125826.125932

Baer, J.-L. and Chen, T.-F., "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE, 1991, pp. 176–186.

Baier, J. L. and Sager, G. R., "Dynamic improvement of locality in virtual memory systems," *IEEE Trans. Softw. Eng.*, vol. 2, no. 1, pp. 54–62, Jan. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233801

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M., "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

Balkesen, C., Teubner, J., Alonso, G., and Özsu, M. T., "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 362–373.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

Callahan, D., Kennedy, K., and Porterfield, A., "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 40–52.

Cantin, J. F., Lipasti, M. H., and Smith, J. E., "Stealth prefetching," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 274–282.

Cavus, M., Karsli, I. B., and Sendag, R., "Array tracking prefetcher for indirect accesses," 2015.

Cavus, M., Sendag, R., and J. Yi, J., "Array tracking prefetcher for indirect accesses," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 132–139.

Changhee Jung, Daeseob Lim, Jaejin Lee, and Solihin, Y., "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, April 2006, pp. 10 pp.–.

Chappell, R. S., Tseng, F., Yoaz, A., and Patt, Y. N., "Microarchitectural support for precomputation microthreads," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 74–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=774861.774870

Charney, M. J. and Reeves, A. P., "Generalized correlation based hardware prefetching," 1995.

Charney, M. J., "Correlation-based hardware prefetching," Ph.D. dissertation, Ithaca, NY, USA, 1995, uMI Order No. GAX95-42429.

Chen, S., Ailamaki, A., Gibbons, P. B., and Mowry, T. C., "Improving hash join performance through prefetching," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 3, p. 17, 2007.

Chen, T.-F. and Baer, J.-L., "Effective hardware-based data prefetching for high-performance processors," *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

Chiba, N. and Nishizeki, T., "Arboricity and subgraph listing algorithms," *SIAM Journal on computing*, vol. 14, no. 1, pp. 210–223, 1985.

Chilimbi, T. M. and Hirzel, M., "Dynamic hot data stream prefetching for general-purpose programs," in *ACM SIGPLAN Notices*, vol. 37, no. 5. ACM, 2002, pp. 199–209.

Chilimbi, T. M. and Hirzel, M., "Dynamic hot data stream prefetching for general-purpose programs," *SIGPLAN Not.*, vol. 37, no. 5, pp. 199–209, May 2002. [Online]. Available: http://doi.acm.org/10.1145/543552.512554

Chou, Y., "Low-cost epoch-based correlation prefetching for commercial applications," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 301–313.

Collins, J., Sair, S., Calder, B., and Tullsen, D. M., "Pointer cache assisted prefetching," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 2002, pp. 62–73.

Collins, J. D., Tullsen, D. M., Wang, H., and Shen, J. P., "Dynamic speculative precomputation," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 306–317.

Collins, J. D., Wang, H., Tullsen, D. M., Hughes, C., Lee, Y.-F., Lavery, D., and Shen, J. P., "Speculative precomputation: Long-range prefetching of delinquent loads," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 14–25, May 2001. [Online]. Available: http://doi.acm.org/10.1145/384285.379248

Cooksey, R., Jourdan, S., and Grunwald, D., "A stateless, content-directed data prefetching mechanism," in *ACM SIGPLAN Notices*, vol. 37, no. 10. ACM, 2002, pp. 279–290.

Cooksey, R., Jourdan, S., and Grunwald, D., "A stateless, content-directed data prefetching mechanism," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 279–290, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635506.605427

D. Collins, J., Sair, S., Calder, B., and Tullsen, D., "Pointer cache assisted prefetching," vol. 2002, 01 2002, pp. 62–73.

Dahlgren, F. and Stenstrom, P., "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, Jan 1995, pp. 68–77.

Dahlgren, F., Dubois, M., and Stenstrom, P., "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *1993 International Conference on Parallel Processing-ICPP'93*, vol. 1. IEEE, 1993, pp. 56–63.

Dundas, J. and Mudge, T., "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 68–75. [Online]. Available: http://doi.acm.org/10.1145/263580.263597

Ebrahimi, E., Mutlu, O., and Patt, Y. N., "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 7–17.

Ebrahimi, E., Mutlu, O., and Patt, Y. N., "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 7–17.

Emma, P. G., Hartstein, A., Puzak, T. R., and Srinivasan, V., "Exploring the limits of prefetching," *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 127–144, 2005.

Ferdman, M., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A., "Temporal instruction fetch streaming," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 1–10.

Fu, J. W., Patel, J. H., and Janssens, B. L., "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

Ganusov, I. and Burtscher, M., "Future execution: A hardware prefetching technique for chip multiprocessors," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 350–360. [Online]. Available: http://dx.doi.org/10.1109/PACT.2005.23

Ganusov, I. and Burtscher, M., "Future execution: A prefetching mechanism that uses multiple cores to speed up single threads," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 4, pp. 424–449, Dec. 2006. [Online]. Available: http://doi.acm.org/10.1145/1187976.1187979

Group, I. R. *et al.*, "Parboil benchmark suite," 2007.

Hashemi, M., Mutlu, O., and Patt, Y. N., "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 61.

Hilton, A., Nagarakatte, S., and Roth, A., "icfp: Tolerating all-level cache misses in in-order processors," *IEEE Micro*, vol. 30, no. 1, pp. 12–19, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.20

Hur, I. and Lin, C., "Memory prefetching using adaptive stream detection," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 397–408.

Hur, I. and Lin, C., "Memory prefetching using adaptive stream detection," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 397–408.

Ishii, Y., Inaba, M., and Hiraki, K., "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 499–500. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542349

Ishii, Y., Inaba, M., and Hiraki, K., "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 499–500.

Ishii, Y., Inaba, M., and Hiraki, K., "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, no. 2011, pp. 1–24, 2011.

Jain, A. and Lin, C., "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 247–259.

Joseph, D. and Grunwald, D., "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 252–263.

Joseph, D. and Grunwald, D., "Prefetching using markov predictors," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 252–263, May 1997. [Online]. Available: http://doi.acm.org/10.1145/384286.264207

Joseph, D. and Grunwald, D., "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 252–263. [Online]. Available: http://doi.acm.org/10.1145/264107.264207

Joseph, D. and Grunwald, D., "Prefetching using markov predictors," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 121–133, Feb. 1999. [Online]. Available: https://doi.org/10.1109/12.752653

Jouppi, N. P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 364–373. [Online]. Available: http://doi.acm.org/10.1145/325164.325162

Jouppi, N. P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 364–373.

Kamruzzaman, M., Swanson, S., and Tullsen, D. M., "Inter-core prefetching for multicore processors using migrating helper threads," *SIGPLAN Not.*, vol. 46, no. 3, pp. 393–404, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950411

Kocberber, O., Falsafi, B., and Grot, B., "Asynchronous memory access chaining," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, 2015.

Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., and Ranganathan, P., "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540748

Lee, J., Kim, H., and Vuduc, R., "When prefetching works, when it doesnt, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, p. 2, 2012.

Lipasti, M. H., Schmidt, W. J., Kunkel, S. R., and Roediger, R. R., "Spaid: Software prefetching in pointer-and call-intensive environments," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press, 1995, pp. 231–236.

Luk, C.-K., "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 40–51.

Luk, C.-K. and Mowry, T. C., "Compiler-based prefetching for recursive data structures," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 222–233, Sept. 1996. [Online]. Available: http://doi.acm.org/10.1145/248208.237190

Luk, C.-K. and Mowry, T. C., "Compiler-based prefetching for recursive data structures," in *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5. ACM, 1996, pp. 222–233.

Michaud, P., "A best-offset prefetcher," in *2nd Data Prefetching Championship*, 2015.

Mowry, T. C., "Tolerating latency through software-controlled data prefetching," Ph.D. dissertation, to the Department of Electrical Engineering.Stanford University, 1994.

Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A., "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N., "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, Feb 2003, pp. 129–140.

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N., "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, Nov 2003.

Mutlu, O., Kim, H., and N. Patt, Y., "Efficient runahead execution: Power-efficient memory latency tolerance," *Micro, IEEE*, vol. 26, pp. 10 – 20, 02 2006.

Nesbit, K. J. and Smith, J. E., "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, Feb 2004, pp. 96–96.

Nesbit, K. J., Dhodapkar, A. S., and Smith, J. E., "Ac/dc: An adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004, pp. 135–145.

Nesbit, K. J. and Smith, J. E., "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 96–96.

Onur Mutlu, Hyesoon Kim, and Patt, Y. N., "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, June 2005, pp. 370–381.

Page, L., Brin, S., Motwani, R., and Winograd, T., "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

Palacharla, S. and Kessler, R. E., "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

Peled, L., Mannor, S., Weiser, U., and Etsion, Y., "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

Pugsley, S. H., Chishti, Z., Wilkerson, C., Chuang, P.-f., Scott, R. L., Jaleel, A., Lu, S.-L., Chow, K., and Balasubramonian, R., "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 626–637.

Roth, A. and Sohi, G. S., "Speculative data-driven multithreading," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001, pp. 37–48.

Roth, A., Moshovos, A., and Sohi, G. S., "Dependence based prefetching for linked data structures," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 115–126, 1998.

Roth, A., Moshovos, A., and Sohi, G. S., "Dependence based prefetching for linked data structures," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 115–126, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/384265.291034

Roth, A. and Sohi, G. S., "Effective jump-pointer prefetching for linked data structures," in *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2. IEEE Computer Society, 1999, pp. 111–121.

Sair, S., Sherwood, T., and Calder, B., "A decoupled predictor-directed stream prefetching architecture," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 260–276, March 2003.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

Shevgoor, M., Koladiya, S., Balasubramonian, R., Wilkerson, C., Pugsley, S. H., and Chishti, Z., "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 141–152.

Sklenář, I., "Prefetch unit for vector operations on scalar computers," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 4, pp. 31–37, 1992.

Smith, A. J., "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec 1978.

Smith, A. J., "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

So, K. and Rechtschaffen, R. N., "Cache operations by mru change," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.

Solihin, Y., Jung, C., Lim, D., and Lee, J., "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 09, pp. 1309–1324, sep 2009.

Solihin, Y., Lee, J., and Torrellas, J., "Using a user-level memory thread for correlation prefetching," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 171–182.

Somogyi, S., Wenisch, T. F., Ailamaki, A., and Falsafi, B., "Spatio-temporal memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 69–80.

Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A., "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 252–263.

Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N., "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 63–74.

Srinivasan, S. T., Rajwar, R., Akkary, H., Gandhi, A., and Upton, M., "Continual flow pipelines," *SIGOPS Oper. Syst. Rev.*, vol. 38, no. 5, pp. 107–119, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1037949.1024407

Srinivasan, V., Davidson, E. S., and Tyson, G. S., "A prefetch taxonomy," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 126–140, 2004.

Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C., "Guided region prefetching: a cooperative hardware/software approach," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE, 2003, pp. 388–398.

Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A., "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 99–108.

Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A., "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 79–90.

Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A., "Mechanisms for store-wait-free multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 266–277, 2007.

Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A., "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 99–108.

Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A., "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 79–90.

Wenisch, T. F., Somogyi, S., Hardavellas, N., Kim, J., Ailamaki, A., and Falsafi, B., "Temporal streaming of shared memory," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 222–233, 2005.

Wenisch, T. F., Somogyi, S., Hardavellas, N., Kim, J., Ailamaki, A., and Falsafi, B., "Temporal streaming of shared memory," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 222–233, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069989

Wu, Y., Serrano, M., Krishnaiyer, R., Li, W., and Fang, J., "Value-profile guided stride prefetching for irregular code," in *International Conference on Compiler Construction*. Springer, 2002, pp. 307–324.

Wulf, W. A. and McKee, S. A., "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

Yang, C.-L. and Lebeck, A. R., "Push vs. pull: Data movement for linked data structures," in *Proceedings of the 14th international conference on Supercomputing*. ACM, 2000, pp. 176–186.

Young, V. and Krishna, A., "Towards bandwidth-efficient prefetching with slim ampm," *The 2nd Data Prefetching Championship*, 2015.

Yu, X., Hughes, C. J., Satish, N., and Devadas, S., "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 178–190.

Zhang, W., Tullsen, D. M., and Calder, B., "Accelerating and adapting precomputation threads for effcient prefetching," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–95. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346187

Zilles, C. and Sohi, G., "Execution-based prediction using speculative slices," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 2–13, 2001.